INDICE

Calcolatore? Non solo ...

Nel nostro corso useremo spesso la parola computer (calcolatore). Ed anche se questa è entrata ormai a far parte del linguaggio comune, è assai riduttiva. Questo termine sarebbe infatti appropriato per quelle "macchinette" che tenete negli astucci e che sono in grado di svolgere giusto le quattro operazioni elementari, l'estrazione di radice e poco più (sto volutamente ignorando le cosiddette calcolatrici programmabili che, di fatto, sono dei personal computer in miniatura, anche se con funzionalità limitate).

Si dovrebbe infatti parlare di sistema di elaborazione elettronico delle informazioni programmabile.

Soffermiamoci su ogni termine:

- Sistema Insieme di componenti, ognuno con la sua specifica funzione, ma con uno scopo comune: elaborare (si parla di 'processing') dati forniti in ingresso (si parla di 'input') e fornire risultati (si parla di 'output') adeguatamente presentati.

  In un moderno sistema di elaborazione elettronico possiamo individuare tra i componenti: il microprocessore (ad esempio Intel Pentium, AMD Athlon), la memoria di lavoro RAM, il disco fisso (hard disk), il monitor, la stampante ecc.

- Elaborazione Operazione (tra cui i calcoli) che trasforma uno o più dati/informazioni in altri dati/informazioni. Questi dati possono insomma essere sì tra loro sommati, sottratti ecc. (se sono numeri); ma possono anche essere confrontati tra loro, spostati o copiati da un punto all'altro della memoria, inviati ad un dispositivo per la loro visualizzazione (ad esempio il monitor) o stampa.

> NOTA: un dato è una misurazione di un aspetto della realtà e diventa informazione solo quando sappiamo dare un significato ad esso (ecco allora che un numero da anonimo diventa un peso, un'altezza, un punto di un'immagine sul video ecc.

- Elettronico Un'elaborazione può avvenire anche in modo manuale (come quando con carta e penna si mette in ordine alfabetico un elenco di nomi).

  Se l'elaborazione avviene senza l'intervento umano si parla allora di elaborazione automatica (ad esempio le macchine per lo smistamento della posta).

  E se infine i dispositivi automatici non hanno parti meccaniche movimento ma sono costituiti da circuiti elettrici si parla di elaborazione elettronica (il microprocessore che somma due numeri).

- Programmabile Con le macchinette calcolatrici non potrete fare altro che i calcoli previsti dal costruttore. Non c'è modo infatti di istruire quel piccolo congegno a svolgere calcoli diversi. Allo stesso modo in cui non potete ottenere altro da una lavatrice che le sequenze (programmi) di lavaggio previste dal costruttore! E così come con una lavatrice potrete lavare solo panni, allo stesso modo con una calcolatrice potrete usare solo numeri!

  Un computer, invece, è dotato di una memoria di lavoro elettronica (RAM, Random Access Memory, Memoria ad accesso casuale) in cui possono essere rappresentati numeri, lettere, immagini e suoni.

  Non solo: la memoria contiene anche la sequenza delle istruzioni che il microprocessore deve eseguire per svolgere un certo compito (il programma)! È sufficiente caricare (dall'hard disk, dal CD, dal DVD ecc.) una diversa sequenza di istruzioni per avere una macchina elettronica in grado di svolgere un compito anche completamente diverso dal precedente.

  Tutto sommato, il ciclo di funzionamento di un sistema di elaborazione elettronico programmabile (ehm, computer d'ora in avanti e solo per comodità) è assai semplice: prelievo dalla RAM dalla prossima istruzione da eseguire, interpretazione dell'istruzione (cosa deve essere fatto? Quali altri componenti devono essere attivati ed in che modo?) esecuzione (attivare nella giusta sequenza i componenti elettronici coinvolti). Si parla di ciclo di fetch (prelievo), decode (decodifica) ed execute (esecuzione).

NOTA: approfondirete questi argomenti nel corso parallelo di 'sistemi'

Logica cablata e logica programmata

Quando un dispositivo elettronico viene costruito in modo da poter funzionare senza un programma (i circuiti sono scelti e collegati per dare sempre la stessa gamma di risposte) viene definito in logica cablata (dalla parola inglese cable, cavo, filo).

Un altro esempio (avevamo già visto quello della lavatrice) è rappresentato da un orologio digitale di prima generazione (che differenza della lavatrice non presenta parti elettromeccaniche). In un orologio di questo tipo non è presente alcun microprocessore e nessun programma: i circuiti sono stati stampati per reagire in modo prefissato alla pressione dei tasti (reset, regolazione orario/data ecc.) e per incrementare l'orario/data.

O ancora: una PlayStation costruita (per assurdo) in logica cablata vi consentirebbe di giocare a quel solo gioco corrispondente a quella particolare predisposizione di dispositivi elettronici e relativi collegamenti elettrici.

Quando invece è presente un microprocessore che esegue istruzioni prelevate da una memoria si parla di logica programmata. Spesso la memoria può essere riscritta (come avviene in tutti i personal computer) e il programma è cambiato piacere. Ecco allora che il computer può di volta in volta diventare l'equivalente elettronico di una macchina da scrivere (Word), di una super calcolatrice (Excel), simulatore di calcio (FIFA e simili!) ecc.

E le soluzioni che sfruttano una logica cablata vengono dette soluzioni hardware (dove con questo termine si indicano le parti fisiche di un sistema di elaborazione; in inglese il termine significa letteralmente ' ferraglia'). Quelle che sfruttano invece una logica programmata sono dette soluzioni software (i dati e le istruzioni memorizzate sottoforma di un segnale elettrico chiaramente impalpabile, morbido, soft (che significa appunto ' soffice').

La soluzione software è più flessibile: se viene trovato un errore è sufficiente cancellare qualche istruzione e la memoria e sostituirle con quelle corrette. Lo stesso accade nel caso si decida di apportare migliorie o adattare il programma a causa, per esempio, di una legge cambiata. In un circuito, invece, se viene trovato un errore questo può comportare lo scarto dell'intero circuito stesso e la realizzazione di un circuito completamente nuovo!

La rigidità delle soluzioni hardware sembrano relegare queste ultime ad un ruolo di secondo piano rispetto alle soluzioni software, ma hanno almeno un grosso pregio che potrebbe essere determinante: la rapidità nel fornire la risposta! Mancando infatti il microprocessore e non essendoci codici d'istruzione da prelevare nella RAM né decodifiche da effettuare, i dati in input sono trasformati in quelli di output ad una velocità molto superiore rispetto ad un programma che compie la stessa elaborazione.

Nel nostro corso ci occuperemo di trovare soluzioni software ai problemi che affronteremo. E l'attività di programmazione non è l'unica da mettere in gioco per risolvere un problema.

Dal problema al programma – ciclo di sviluppo (semplificato) del software

La scrittura del programma è solo una delle fasi del processo di sviluppo di un'applicazione informatica. Tutto inizia con l'esigenza di risolvere un problema con un sistema informatico. La parola problema deve essere intesa in modo ampio: gestire la contabilità di un'azienda, usare il personal computer per scrivere documenti, per una simulazione di guida, per svolgere calcoli complessi, controllare un processo industriale, pilotare un robot eccetera.

Per quanto ci riguarda, i problemi saranno espressi in forma testuale. Ecco il testo dell'esame di Stato del 1998:

> Una galleria d'arte ha deciso di creare un sistema che consenta ai suoi clienti di consultare da casa il catalogo: dei quadri, tramite accesso a una pagina web che la galleria può creare presso un fornitore di servizi telematici.
>
> Per ogni quadro è compilata una scheda che riporta l'autore, il titolo, la tecnica (olio, tempera ecc.), le dimensioni, il prezzo. La consultazione del catalogo: può avvenire o semplicemente scorrendo avanti e indietro le schede in ordine alfabetico oppure cercando uno specifico autore.
>
> Il candidato, fatte le ipotesi aggiuntive che ritiene necessarie,
>
> 1) proponga una soluzione per la creazione del sistema illustrandone la struttura a blocchi e indicando una soluzione di principio per ciascun blocco;
>
> 2) proponga e illustri una struttura per il Data Base dei quadri,
>
> 3) sviluppi in dettaglio la soluzione per almeno una delle seguenti funzioni, codificandone un segmento con uno strumento software di sua conoscenza:
> a) creazione del Data Base,
> b) creazione di una semplice pagina web della galleria,
> c) interfaccia per consentire al cliente la consultazione del catalogo: e la visione delle singole schede,
>
> 4) facoltativamente proponga una soluzione di principio per realizzare un sistema che consenta di mostrare al cliente non solo la scheda di catalogo, ma anche una fotografia del quadro.

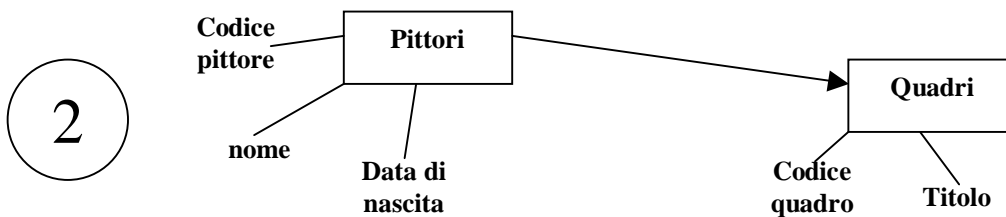
Il testo va dapprima ha studiato per evidenziare parti poco chiare (sulle quali sarà necessario prendere delle decisioni) ed eventualmente fare ipotesi aggiuntive su aspetti per i quali il testo non dice come comportarsi.

> Ad esempio, nel testo si parla di prezzo del quadro ma non viene indicata la valuta da utilizzare. Una galleria d'arte spesso a clienti stranieri, per cui la soluzione di proporre il prezzo in lire (siamo nel '98... e l'euro non esisteva ancora e neppure l'obbligo di esporre il doppio prezzo in euro ed in lire) forse non è ottimale. Ecco allora la prima ipotesi aggiuntiva (o, se preferite, un primo chiarimento): i prezzi verranno indicati sia in lire che in dollari.

Questa fase viene chiamata studio della situazione reale. La letteratura informatica fa riferimento a questa fase anche con il nome di analisi (termine però non corretto da un punto di vista matematico). Naturalmente noi inizieremo con problemi e testi assai più semplici. Ad esempio: calcolare la spesa in euro per una settimana di viaggi andata e ritorno da casa a scuola. Le uniche cose da chiarire potrebbero essere: è necessario percorrere strade con pedaggi? Il numero di chilometri del percorso in andata è identico a quello del ritorno?

Vengono anche presi in considerazione i requisiti (cosa deve fare il programma, con quali vincoli di velocità, occupazione di memoria, hardware e software a disposizione, se deve funzionare in rete, se deve sapere interoperare con altri software magari su piattaforme hardware diverse, se deve funzionare in real time, se deve essere portabile in altri ambienti hardware/software, , che grado di robustezza, che grado di sicurezza, che tipo di periferiche deve supporatare ecc.)

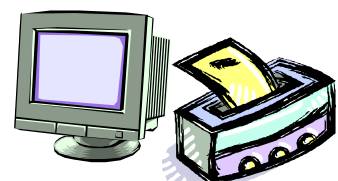
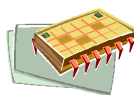
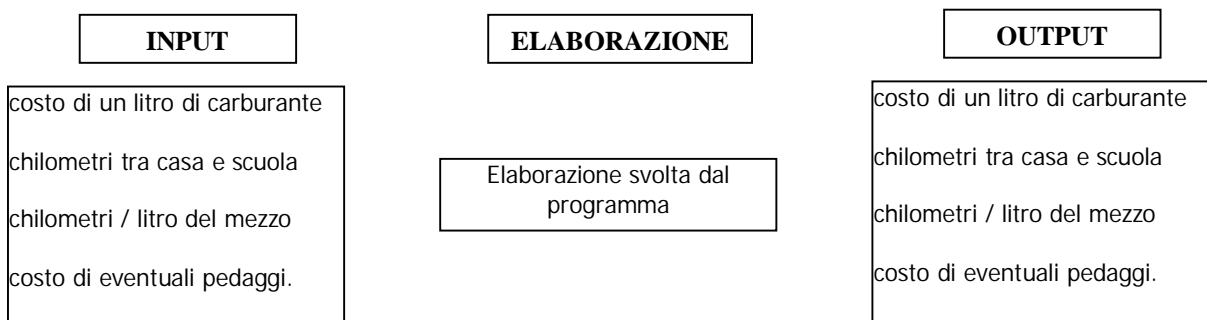
Devono poi essere individuate tutte le informazioni che è necessario gestire. Queste spesso verranno memorizzate in una banca dati (data base). Il risultato di questa fase viene di solito sintetizzato con uno schema che evidenzia i cosiddetti insiemi entità e le relazioni tra esse.

Ad esempio, è possibile individuare l'insieme entità dei quadri e quello dei pittori:

Per ogni insieme entità vengono anche indicati gli attributi che descrivono un esemplare di quell' insieme. Ad esempio, per l'insieme entità dei pittori si decide che ogni pittore verrà descritto tramite un codice, un nome ed una data di nascita. Ogni quadro verrà invece descritto da un codice e da un titolo. La freccia che da pittori e raggiunge quadri sta ad indicare che per ogni pittore esiste un certo numero di quadri. Nella banca dati verrà creata una tabella 'pittori' che conterrà su ogni sua riga i dati di un pittore e similmente per i quadri. Ogni quadro potrebbe essere associato al suo pittore indicando nel quadro il codice del pittore.

Questa fase viene chiamata analisi dei dati.

Anche per questa fase inizieremo con situazioni molto semplici. Proseguendo con l'esempio di problema presentato al punto uno, non è difficile convincersi che i dati di cui abbiamo bisogno sono: costo di un litro di carburante, numero di chilometri tra la casa e la scuola, numero di chilometri che il mezzo utilizzato compie con un litro di carburante, costo di eventuali pedaggi.

Per queste semplici situazioni uno schema come quello appena visto è esagerato. Ci accontenteremo di elencare i cosiddetti dati in ingresso (input); il programma li elabora (processing) per produrre i risultati, i dati in uscita (output).

INPUT	**ELABORAZIONE**	**OUTPUT**
costo di un litro di carburante chilometri tra casa e scuola chilometri / litro del mezzo costo di eventuali pedaggi.	Elaborazione svolta dal programma	costo di un litro di carburante chilometri tra casa e scuola chilometri / litro del mezzo costo di eventuali pedaggi.

Deve essere molto chiara una cosa: per dati in input si intendono quelli indispensabili, quelli cioè che il computer non può calcolare o derivare in altro modo. Facciamo un esempio: se ad un certo punto in un programma abbiamo disposizione una quantità espressa in ore e per proseguire è necessario esprimerla i secondi, non cediamo alla pigrizia chiedendo a chi sta usando il programma di inserire questo valore usando la tastiera, pretendendo che sia lui a fare la conversione! Sarà invece il programma a calcolare autonomamente il valore richiesto moltiplicando per 3600 il numero delle ore...

NOTA: non è raro il caso di alunni in difficoltà al momento di individuare i dati in input. Intanto diciamo che non è necessario avere la certezza di averli individuati proprio tutti per poter proseguire: è normale, dopo avere individuato i più importanti ed evidenti, iniziare la fase successiva (trovare un 'modo' di utilizzare i dati in input per giungere alla risultato); si ad un certo punto ci si accorge che manca qualche dato per poter proseguire lo si aggiungerà semplicemente ai dati di input.

È arrivato il momento di descrivere il modo in cui i dati di input devono essere utilizzati per ottenere i risultati. Questa 'descrizione' è chiamata algoritmo.

Da 'Wikipedia', l'enciclopedia libera (http://it.wikipedia.org), con qualche piccolo adattamento:

Il termine (algoritmo) deriva dal nome del (grande) matematico arabo Al-Khwarizmiī, che pubblicò, tra gli altri, il libro dal quale prende le origini la parola Algebra (ora sapete chi odiare). Nei suoi libri ne scrive anche i procedimenti per portare a termine alcuni tipi di calcolo: questi procedimenti presero il nome di algoritmi.

Nella sua definizione più semplice ed intuitiva un algoritmo è una sequenza ordinata di passi semplici che hanno lo scopo di portare a termine un compito più complesso (una ricetta da cucina, ad esempio, può essere considerata come un algoritmo che partendo da un insieme di singoli alimenti di base ed eseguendo una sequenza di passi, produce come risultato un piatto composto).

> In un modo più formale, possiamo quindi definire l'algoritmo come una sequenza ordinata e finita di istruzioni che, dato uno od una serie di elementi in input, produce uno od una serie di risultati in output .

Sequenza ordinata significa che esiste un ordine preciso in base al quale vengono eseguite le istruzioni (d'altronde sarebbe ben difficile prima sbattere un uovo e poi rompere il guscio!).

Sequenza finita significa che le istruzioni possono essere anche veramente tante, ma non in numero limitato; inoltre il numero di volte che globalmente queste istruzioni vengono eseguite non può essere illimitato.

La sequenza delle operazioni deve essere chiara, mai ambigua, deve avere un ordine ben preciso, e deve giungere a termine per ogni input. Tutte le istruzioni devono comportare delle azioni tra quelle che l'esecutore è in grado di svolgere. Il risultato di un algoritmo deve essere sempre uguale indipendentemente da chi lo esegue.

Se, come visto, una ricetta da cucina rappresenta un discreto esempio di algoritmo direttamente eseguibile da un essere umano, l'istruzione "aggiungere sale quanto basta" difficilmente sarà comprensibile per una macchina (ma anche tra gli umani stessi quel ' quanto basta' verrebbe sicuramente interpretato in tanti modi diversi!).

Un passo di un algoritmo può essere definito anche tramite un altro algoritmo (chiamato in questo caso sottoalgoritmo), che suddivide il compito in compiti ancora più elementari. Facciamo un esempio: l'algoritmo "va dal salotto alla cucina" si compone in realtà delle seguenti istruzioni:

- esci dal salotto
- curva a sinistra
- prosegui per il corridoio fino all'ultima porta sulla sinistra
- attraversa la porta a sinistra

Questo è certamente fin troppo esplicito per un operatore umano (al quale già il problema originale "va dal salotto alla cucina" sembra probabilmente abbastanza elementare da non richiedere, in apparenza, suddivisioni), ma nel caso di un robot richiederebbe di specificare i passi con ben altra (minore) complessità.

L'algoritmo "attraversa la porta a sinistra" si compone di:

- controlla se la porta è aperta
- nel caso che la porta sia aperta salta il passo seguente
- apri la porta
- avanza di un metro

L'algoritmo "apri la porta", compreso nel precedente, a sua volta si compone di:

- protendi il braccio
- afferra la maniglia
- rotea la mano di 30 gradi in direzione antioraria
- applica una pressione alla maniglia diretta di fronte a te
- ...

Un modo dettagliato di rappresentare l'algoritmo "attraversa la porta a sinistra" è allora il seguente:

- controlla se la porta è aperta
- nel caso che la porta sia aperta salta il passo seguente
- apri la porta
- protendi il braccio
- afferra la maniglia
- rotea la mano di 30 gradi in direzione antioraria
- applica una pressione alla maniglia diretta di fronte a te
- ...
- avanza di un metro

Una breve analisi dell'esempio sopra, porta a delineare alcune caratteristiche essenziali di un algoritmo:

- non ambiguo: le istruzioni devono essere univocamente interpretabili;
- eseguibile: ogni istruzione deve terminare in tempo finito.
- Inoltre, in informatica, si richiede generalmente che un algoritmo sia finito, ovvero termini per ogni insieme di dati di ingresso.

Un algoritmo non è tale se risolve in un caso particolare di un problema: deve essere utile per la soluzione di un'intera classe di problemi. Facciamo un esempio: il procedimento che serve a calcolare l'area del triangolo la cui base misura 3 m e l'altezza 5 m, e solo l'area di questo triangolo, non può definirsi un algoritmo. Il procedimento invece che descrive come calcolare l'area di un qualsiasi triangolo nota la misura della base e dell'altezza, risolve un'intera classe di problemi (è una soluzione generale) e può definirsi algoritmo.

Un programma è la traduzione di un algoritmo in un blocco di istruzioni eseguibili automaticamente da un sistema di elaborazione elettronico.

Arrivati a questo punto dobbiamo confrontarci con l'assoluta inadeguatezza del linguaggio parlato (il cosiddetto linguaggio naturale) per descrivere un algoritmo. Banalizzo con un classico esempio:

la vecchia porta la sbarra

Quale significato deve essere dato questa frase? Si tratta forse di un'anziana signora china sotto il peso di una pesante sbarra? O si sta parlando di uscita sbarrata da una vecchia porta? Questa ambiguità è inaccettabile per un computer: esso deve sapere esattamente come comportarsi e deve produrre sempre gli stessi risultati se gli vengono sottoposti gli stessi dati in input.

È necessario servirsi di linguaggi formali, cioè rigorosamente definiti. Questi tipi di linguaggio sono di solito molto meno ricchi di vocaboli e di regole sintattiche ma hanno il grosso pregio di non essere ambigui (ogni istruzione è chiara, ha un solo significato e produce sempre lo stesso risultato).

Questi linguaggi sono chiamati linguaggi di programmazione. La figura professionale che si occupa della scrittura dei programmi è il programmatore. la fase di scrittura di un programma è detta di codifica (il programmatore scrive il codice del programma)

Non è detto che sia il programmatore a studiare il problema e ad ideare l'algoritmo risolutivo: la figura professionale specializzata in questi compiti preliminari e fondamentali è chiamato analista. E' certamente vero che il ruolo dell'analista e del programmatore possano essere svolti dalla stessa persona. Per compiere una buona analisi è necessaria molta esperienza, ed è per questo che spesso si nasce 'semplici' programmatori per poi diventare analisti o analisti-programmatori.

il ruolo del programmatore ' puro' è quello allora di ricevere dall'analista la descrizione dell'algoritmo per provvedere alla codifica (cioè scrittura) di quest'ultimo usando un linguaggio di programmazione.

> Torneremo presto sulla questione della descrizione degli algoritmi: è il tema portante di quest'anno scolastico!

NOTA: è corretto fare distinzione tra il risolutore di un problema (colui che ha ideato l'algoritmo che lo risolve) e l'esecutore materiale dei passi dell'algoritmo. Nel nostro caso il risolutore è sempre un uomo/donna e l'esecutore è il computer.

5

Terminata la scrittura del programma inizia la fase di test. Sottoporre a test un programma significa provarlo con tutte le configurazioni di dati in input normali e particolari. Di nuovo, facciamo un semplice esempio immaginando di avere scritto un programma che, forniti due numeri in input, calcola che percentuale è il primo rispetto al secondo; ad esempio se il primo numero fosse 50 ed il secondo 150, il risultato fornito dovrebbe essere 33,3 % periodico (50 è infatti un terzo di 150...). Non è difficile convincersi che nel programma la formula usata è:

(primo numero/secondo numero)*100

Fare il test di questo programma con configurazioni di dati in input 'normali', significa provare il programma con coppie di numeri tipo (10,20) (50,150) eccetera. Poi ci si potrebbe domandare se il programma fornisce risultati corretti anche quando il primo numero è maggior dal secondo: (240,60); e scopriremo che la risposta è sì: otterremo come valore 400% (in effetti, 240 è il quadruplo di 60). E se usassimo numeri negativi? Nessun problema...

Ok, è arrivato il momento di essere cattivi: e se usassimo numeri decimali? Tipo (10.2, 97.5) ? E se il primo numero fosse zero: (0, 34)? Anche con queste configurazioni di valori in input il programma continua a fornire risultati corretti. Giunti a questo punto, il programmatore inesperto (o pigro) potrebbe concludere che il programma funziona bene in tutti i casi possibili. Purtroppo, la matematica c'insegna che non è possibile dividere per zero: inserendo una configurazione di input con il secondo uguale a zero, come in (72,0), il programma andrebbe letteralmente tilt! Gli informatici in questi casi usano un'espressione assai colorita: il programma va in crash!

Il caso dello zero come secondo numero è un cosiddetto caso limite: ogni programma dovrebbe essere testato in tutti i casi limite che potrebbero presentarsi, anche se con probabilità molto bassa!

La fase di test viene di solito suddivisa a sua volta in:

- alpha test: è quello svolto direttamente dal programmatore che ha scritto il codice o comunque da personale interno alla ditta che commercializzerà il software; potremmo dire che in questa fase vengono trovati gli errori più grossolani;

- beta test: quando il software viene ritenuto sufficientemente stabile viene distribuito, di solito gratuitamente, ad un numero ristretto di utenti che, in cambio del beneficio di poter disporre in anteprima del prodotto quasi finito, comunicheranno secondo protocolli stabiliti una descrizione degli errori che capitano durante l'utilizzo; se il prodotto è particolarmente complesso, il numero dei beta tester, può essere elevato: ad esempio, quando la Microsoft rilascia per il beta test una nuova versione di Windows lo fa anche a decine di migliaia di utenti!

- gamma test: a volte viene definito un ulteriore livello che si differenzia dal precedente solo del fatto che dovrebbe essere quasi esente da errori

La fase di test ha lo scopo di evidenziare gli errori durante il funzionamento del programma, il che non vuol dire però automaticamente sapere quale istruzione ha causato il problema.

La fase di debug indica invece l'attività del programmatore volta ad individuare esattamente la porzione di codice che contiene l'errore in modo da poter eliminare.

Ogni volta che si scopre un errore è necessario ritornare alla fase di codifica per modificare il codice.

ATTENZIONE!! Una delle cattive abitudini più comuni è quella di non testare nuovamente il programma dopo l'eliminazione di un errore. Purtroppo l'esperienza insegna che, non così raramente come si potrebbe pensare, le modifiche apportate per correggere un errore ne introducono altri!

Finalmente il codice può essere immesso sul mercato! Si parla di rilascio (release). Ma non è finita qui! Anzi, rapportato a 100, il totale dell'impegno rappresentato da queste fasi viene stimato da molti come non superiore al 40%

Ed il resto ? Siate sinceri: quante volte vi è capitato di acquistare un video game senza essere costretti ad applicare una cosiddetta patch (letteralmente pezza, correzione). Quante volte come utenti siamo rimasti in attesa dell'ennesimo service pack (un kit che contiene molte patch 'in un colpo solo') di Windows o di Office o di altri software? Ma anche senza errori, ogni anno vengono immesse sul mercato nuove versioni (qualcuno riesce a contare quelle della serie FIFA? o di Final Fantasy?).

Il 60% che manca di tutto il costo per lo sviluppo del software viene fagocitato dalla cosiddetta manutenzione. Ci sono tre tipi di manutenzione:

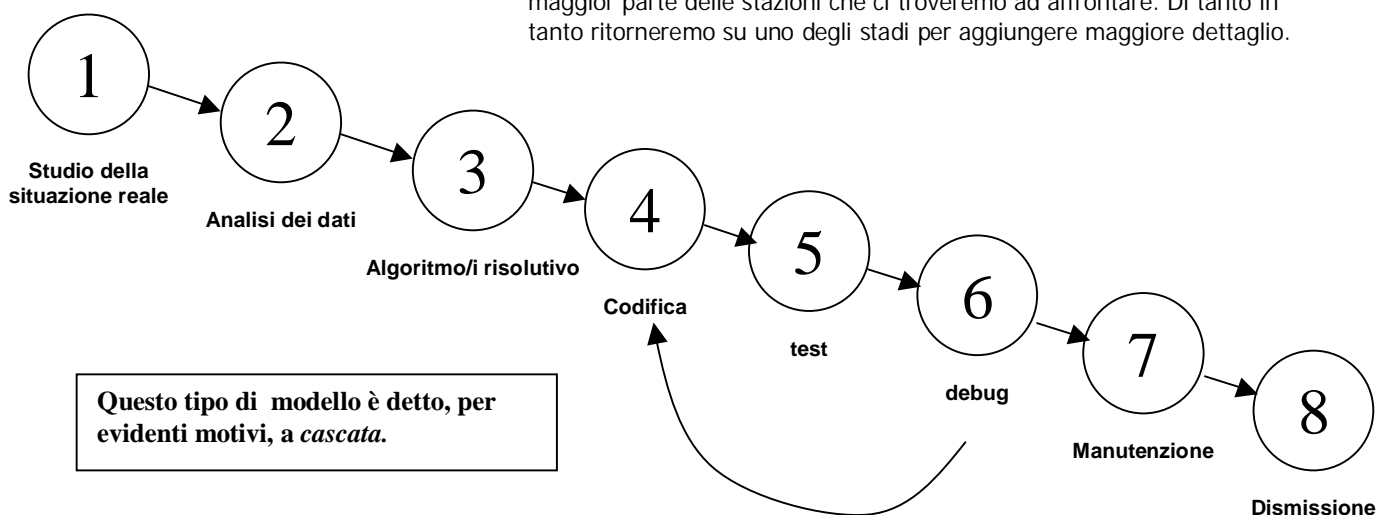
- correttiva: anche dopo aver superato tutte le fasi di test, è quasi impossibile che un software non contenga ancora almeno un errore; periodicamente, quindi, vengono rilasciate versioni che si spera siano di volta in volta meno affette da errori;

- perfettiva o migliorativa: indica tutti quegli interventi che non servono a togliere degli errori ma a migliorare in qualche modo il prodotto; ad esempio a renderlo più veloce, a diminuire le sue esigenze di spazio sul disco, a renderlo capace di riconoscere nuove periferiche, a compiere funzioni prima non previste eccetera;
- adattiva o adattativi: nessun errore da togliere, nessun miglioramento da portare; ma un cambiamento nel contesto in cui il software deve funzionare costringe ad apportare delle modifiche: il caso classico che si cita in questa situazione è la modifica di una legge che forza la software house a rispettarla, a costo di pesanti modifiche del codice!

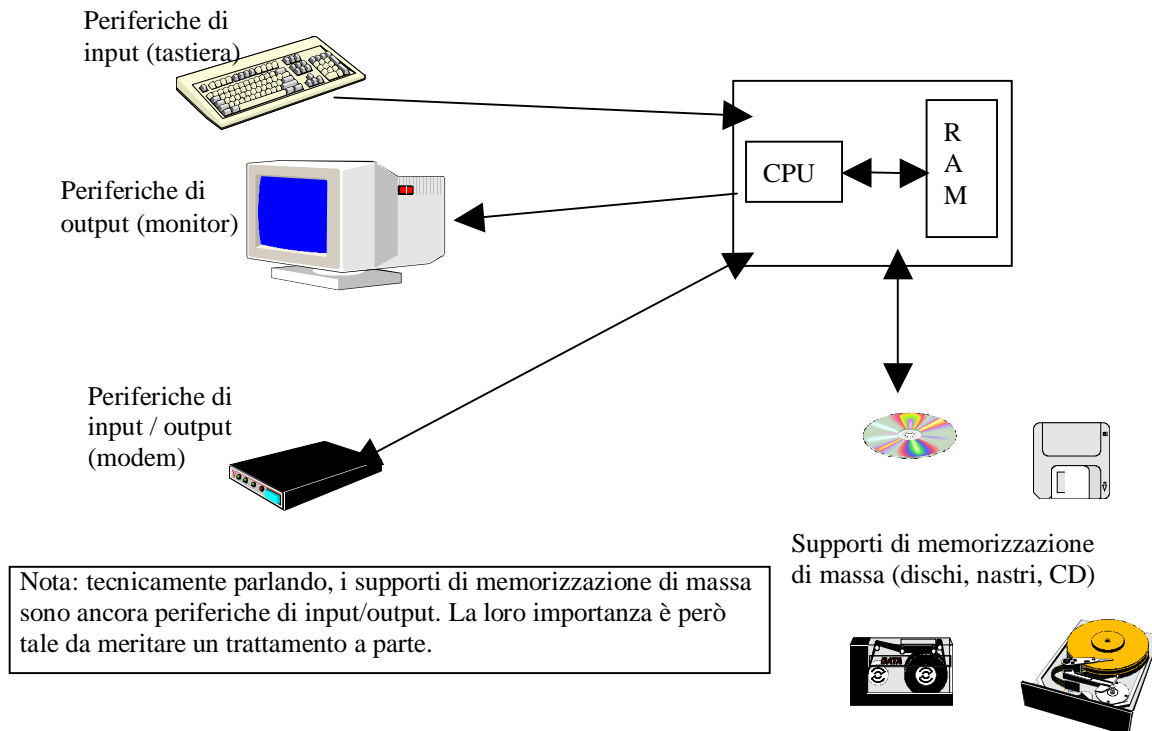
Infine, quando un'applicazione non è più utile (perché superata da altre, perché le modifiche richieste sono troppe, perché un evento imprevisto la rende obsoleta eccetera) se ne può anche decretare la morte, la dismissione.

E' doveroso sottolineare che questo processo nel quale io ho individuato otto stadi non è l'unico modello riconosciuto per lo sviluppo del software (e a dire la verità l'ho semplificato rispetto quelli che trovate nella letteratura informatica). È però facile da capire e si adatta bene alla maggior parte delle stazioni che ci troveremo ad affrontare. Di tanto in tanto ritorneremo su uno degli stadi per aggiungere maggiore dettaglio.

Riassumendo:

Questo tipo di modello è detto, per evidenti motivi, a *cascata*.

Architettura hardware e software di un sistema di elaborazione

Per capire cosa sia un linguaggio di programmazione e come utilizzarlo per scrivere programmi, è prima necessaria una panoramica che mostri a grandi linee il funzionamento di un sistema di elaborazione.

SCHEMA A BLOCCHI DI UN SISTEMA DI ELABORAZIONE

Il cuore del sistema è la CPU (Central Processing Unit, Unità centrale di processo). Nei personal computer la CPU è rappresentata da un singolo chip (il microprocessore): Pentium, Celeron, Athlon, Sempron, PowerPC, non vi dicono nulla questi nomi??

La CPU è il 'cervello' del sistema. Il software di base (il BIOS che controlla direttamente l'hardware, windows/linux cioè i sistemi operativi, gli strumenti di programmazione ed altre utilità di sistema) ed anche quello applicativo (programmi di video scrittura, contabilità, giochi ecc.) è formato da istruzioni, molte istruzioni. Per avere un idea della complessità, pensate che un software applicativo come Word è formato da milioni di istruzioni … Queste indicano ciò che va fatto per consentire all'utente di scrivere un documento usando il computer. Ecco, sinteticamente, come funziona l'intero meccanismo:

Usando l'interfaccia che il sistema operativo mette a disposizione, l'utente comanda il caricamento delle istruzioni (il programma) dai supporti di memorizzazione di massa: nel caso di Windows si fa doppio click, ad esempio, sull'icona di Word …).

Le istruzioni (se non tutte almeno quelle che servono in un primo momento) sono trasferite dai lenti supporti di memorizzazione di massa nella velocissima memoria elettronica (RAM): da quest'ultima saranno a disposizione della CPU in tempi molto brevi.

La CPU preleva la prima istruzione del programma dalla RAM (Random Access Memory, memoria ad accesso casuale).

La CPU 'capisce' cosa gli chiede l'istruzione.

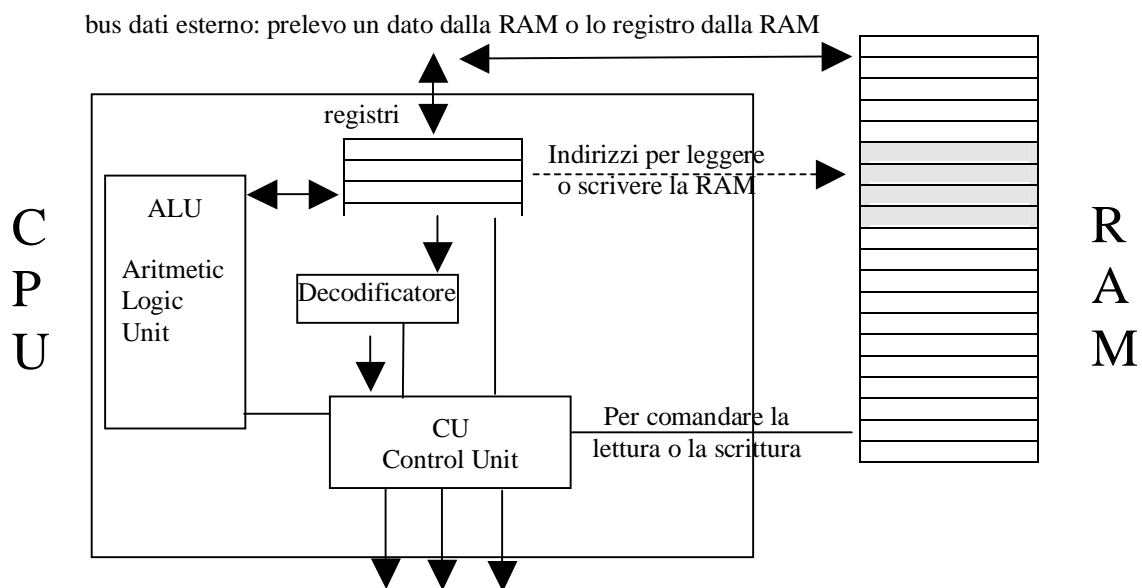
LA CPU invia i necessari comandi a tutti i dispositivi interessati da quell'istruzione, eseguendola.

Se l'istruzione non è quella finale, si ritorna al punto 3 prelevando la successiva istruzione e così via …

Durante l'esecuzione delle istruzioni, la CPU può usare la parte della RAM rimasta libera per registrare dati intermedi: è sempre per motivi di velocità che si preferisce la memoria elettronica a quella di massa (naturalmente si dovrà prima o poi provvedere anche alla registrazione dei dati sui supporti di massa, permanenti, pena la perdita dei dati stessi allo spegnimento del computer). Ad esempio, i documenti creati con Word sono prima registrati nella RAM e solo con un comando esplicito di registrazione (File / Salva) sono memorizzati sui dischi.

Alcune istruzioni chiederanno alla CPU di controllare l'uso di periferiche di input o di output. Sempre pensando a Word, non è difficile convincersi che le sue istruzioni debbano fare in modo che la CPU rimanga in attesa fino a che chi sta usando il programma non preme un tasto qualsiasi o usa il mouse. Non appena l'utente preme un tasto, le istruzioni comandano la sua visualizzazione sul video; se con il mouse viene scelto un comando dal menu, le istruzioni chiederanno alla CPU di comportarsi in modo appropriato.

In realtà tutto questo avviene con la collaborazione del sistema operativo (Windows o Lilnux, ad esempio): quest'ultimo mette a disposizione di ogni programma una serie di servizi che possono essere invocati; ad esempio se un carattere deve essere fatto apparire sullo schermo, tutte le operazioni coinvolte non sono gestite al massimo dettaglio dal programmatore di word (per fortuna!); quello che accade è che il programmatore che ha scritto Word 'chiede' a Windows di fare questo per lui. Sono tantissimi i servizi messi a disposizione da un sistema operativo (gestione di tutte le periferiche, gestione della ram, del file system, della rete ecc.). Quando i sistemi operativi non esistevano ancora od erano molto primitivi, la vita del programmatore era molto più dura (doveva programmare TUTTI i dettagli). Approfondirete questi aspetti nel corso di sistemi.

Esaminiamo ora in dettaglio il funzionamento di una CPU. Ecco un ingrandimento:

Non spaventatevi: è più semplice di quello che sembra a colpo d'occhio. Nella zona in grigio della RAM immaginiamo essere presenti un blocco di istruzioni da eseguire (potrebbero essere quelle di un programma come Word).

La RAM: pensiamola come una sequenza di celle, chiamate byte. In ogni byte può essere memorizzato un dato, come un istruzione od un carattere (vedremo dopo come ed in che forma). Nel disegno qui sopra sono stati evidenziati in grigio quattro byte. Ogni cella (byte) viene indicata con il suo indirizzo, cioè la posizione a partire dall'inizio. Visto che si comincia a contare da zero, quest'ultimo è l'indirizzo della prima cella in alto nella RAM (si parla anche di byte zero). La seconda cella dall'alto ha allora indirizzo uno e così via… I byte in grigio iniziano all'indirizzo cinque e terminano quattro byte più avanti, all'indirizzo otto (considerando il cinque sono appunto quattro indirizzi).

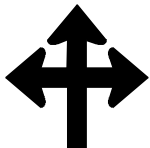
I registri: sono celle di memoria speciali, interne alla CPU, usate per memorizzare valori per diversi scopi: la CPU impiega pochissimo a trovare o scrivere un dato nei registri, perché sono molto vicini ad essa e molto veloci. E' quindi assai conveniente tenere qui i risultati intermedi delle operazioni e tutto ciò che si dovrebbe continuamente rileggere o scrivere nella più lenta RAM. Purtroppo la spazio all'interno della CPU è veramente minimo: c'è spazio solo per alcune decine di registri ... Chiaro che di volta in volta si terrà, della RAM, solo ciò che serve alle istruzioni del momento, poi si dovrà sostituire il contenuto dei registri con un altra parte della RAM. Pur con queste limitazioni, lavorare con i registri dà grossi benefici rispetto all'uso della sola RAM.

Alcuni registri hanno poi una funzione speciale: uno, chiamato contatore di programma (program counter) indica a quale indirizzo della RAM si trova il dato che rappresenta la prossima istruzione da eseguire.

Un altro è chiamato registro indirizzi: è qui che va depositato un indirizzo ogni volta che si vuole leggere o scrivere un byte della RAM (quando si vuole leggere il codice della prossima istruzione da eseguire, il contatore di programma viene infatti copiato qui).

Un altro è chiamato registro istruzioni e contiene il codice numerico dell'istruzione letta nella RAM, che deve essere eseguita (infatti ogni possibile istruzione è rappresentata da un codice numerico).

ALU: è l'Unità Aritmetico Logica. Essa svolge le operazioni aritmetiche (matematiche) elementari ed i confronti logici tra due dati (sa dire se sono uguali, se uno è più grande o più piccolo dell'altro ecc.).

Control Unit: è l'unità di controllo. E' lei che invia a tutti gli altri dispositivi segnali elettrici con cui comanda tutte le micro operazioni che possono essere svolte. Ad esempio può comandare l'invio del contenuto di un registro in un altro registro, oppure la lettura di un byte della RAM ed il trasferimento del suo contenuto in un registro (o il contrario), il trasferimento alla ALU di due dati da sommare o da sottrarre (ed il comando che fa eseguire la somma e la sottrazione). Può inviare segnali di comando anche a dispositivi esterni alla CPU: ad esempio può comandare allo scanner di inviare il prossimo blocco di dati del documento che sta leggendo. Insomma è il vero regista che regola il funzionamento di tutti gli altri dispositivi interni od esterni.

Bus: si è parlato di invio di comandi dalla control unit, di trasportare byte dalla RAM ai registri e viceversa, di comunicare indirizzi per usare la RAM. Come viaggiano tutte queste informazioni ? Semplice, usano il bus ! Battute a parte, il termine indica i canali fisici lungo i quali si spostano i segnali elettrici che rappresentano dati, indirizzi e comandi. Ci sono tre tipi di bus:

Bus dati: come suggerisce il nome, trasporta dati (il contenuto di una cella di RAM, di un registro o un byte da o verso una periferica. Esiste un bus dati interno, sul quale viaggiano i dati all'interno della CPU, ed uno esterno che permette lo scambio dei dati tra la CPU e dispositivi esterni (e viceversa) o tra dispositivi esterni stessi. Ho rappresentato i bus dati con una linea a tratti piccoli. Per non appesantire il disegno non ho collegato tutti i dispositivi che si possono scambiare dati grazie al bus interno. Le frecce indicano naturalmente il senso di percorrenza.

Nelle moderne architetture esistono bus specializzati per scambiare dati con periferiche che consumano questi ultimi ad un tasso elevatissimo. Un esempio è rappresentato dalla scheda grafica: essa sfrutta bus dedicati (connessione AGP, oggi in via di sostituzione con i collegamenti Serial Ata).

Bus indirizzi: ogni volta che la CPU legge o scrive un dato dalla RAM deve prima impostare il relativo indirizzo ; pensate al bus indirizzi come ad un bus dati specializzato nel trasportare indirizzi. Ho indicato il bus indirizzi con una linea a tratti larghi.

Bus controlli: è l'insieme delle linee su cui viaggiano i comandi (segnali elettrici) che la Control Unit invia a dispositivi esterni; serve anche ai dispositivi esterni per segnalare eventi particolari alla CPU: avete presente cosa accade quando finisce la carta della stampante ? O quando il disco che tentate di scrivere è protetto ? O quando non c'è più spazio sul disco ? L'ho indicato nel disegno con le tre linee continue che puntano verso il basso. La control Unit può attivare o disattivare anche i componenti interni grazie a delle linee di selezione: le ho indicate con delle linee continue senza frecce: quando la Control Unit vuole attivare un dispositivo, invia lungo una linea di selezione un segnale apposito.

Ora che sono stati descritti tutti i componenti, vediamoli all'opera. Immaginiamo che le istruzioni da eseguire siano contenute nei byte in grigio:

Il registro program counter, abbiamo detto, contiene l'indirizzo dell'istruzione da eseguire (in questo caso l'indirizzo del primo byte in grigio). Per poterla leggere dalla RAM è necessario che tale indirizzo sia contenuto nel registro indirizzi. L'unità di controllo comanda allora la copia del contenuto del program counter in questo registro.

L'unità di controllo comanda quindi al dispositivo di lettura/scrittura della RAM (non evidenziato nel disegno per semplicità) di leggere il byte all'indirizzo contenuto ora nel registro indirizzi; il suo valore (il codice dell'istruzione) viene trasferito lungo il bus dati esterno e da lì sul bus dati interno, fino a depositarlo nel registro istruzioni.

L'unità di controllo comanda a questo punto al decodificatore di leggere dal registro istruzioni il codice numerico dell'istruzione e di 'capire' che cosa è richiesto.

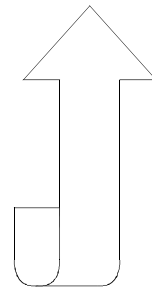
Sulla base della 'risposta' del decodificatore l'unità di controllo attiva in sequenza i dispositivi interessati dall'istruzione; ad esempio (semplificando) se si dovesse calcolare A = B + C

prelevare B dalla RAM (indirizzo di B in registro indirizzi e successiva lettura);

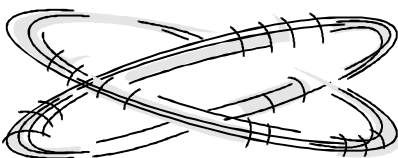
prelevare C dalla RAM (indirizzo di C in registro indirizzi e successiva lettura);

trasferire alla ALU gli operandi e comandare la somma

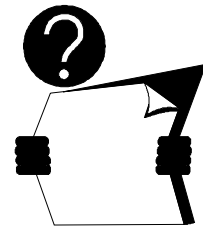
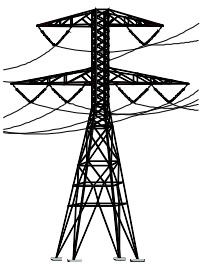
copiare il risultato nella posizione dove in RAM si trova 'C' (indirizzo di C in registro indirizzi e scrittura)

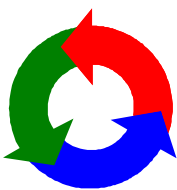
Nel frattempo il program counter è stato automaticamente incrementato e indica l'indirizzo della prossima istruzione da eseguire. Si può allora ricominciare dal punto 1, fino a che ... il computer viene spento ! In realtà, anche dopo che il programma termina, ad esempio chiudendo Word, un altro rimane in funzione: quello che già funzionava prima di Word, Windows in persona. Infatti il sistema operativo, esso stesso un insieme di programmi, è all'opera dal momento dell'accensione del computer fino allo spegnimento. Quando l'utente comanda, usando l'interfaccia grafica, la partenza di un'applicazione, le istruzioni di quest'ultima affiancano in RAM quelle del sistema operativo ed il program counter viene fatto 'puntare' alla prima istruzione dell'applicazione, che inizia così ad essere eseguita. A programma terminato si riprende ad eseguire il sistema operativo. Durante il suo funzionamento l'applicazione può richiamare parti del sistema operativo (richiesta di servizi): il punto a cui si era arrivati con l'applicazione viene prima memorizzato, per poter riprendere, dopo che il sistema operativo ha esaudito la richiesta, il programma esattamente dall'istruzione a cui era stato interrotto.

La fase in cui si provvede a prelevare il codice numerico dell'istruzione da eseguire è chiamata fase di fetch (prelievo).

La fase in cui il decodificatore interpreta l'istruzione, determinando le microoperazioni che verranno poi eseguite dall'unità di controllo è chiamata fase di decode (decodifica).

La fase in cui le microoperazioni sono effettivamente portate a termine, attivando nella giusta sequenza i dispositivi interessati è chiamata fase di execute (esecuzione).

Il ciclo fetch, decode ed execute viene ripetuto a velocità incredibili: oggi miliardi di milioni di volte al secondo (1 volta al secondo = 1 Hertz = 1Hz; un milione di volte al secondo = 1 Mega Hertz = 1Mhz; 1Ghz= un miliardo). Le fantastiche capacità di un sistema di elaborazione sono tutte qui: una velocità pazzesca ed una memoria perfetta nell'eseguire istruzioni semplicissime (sposta un byte di qua e mettilo di là ...) ! E' come se un muratore usasse mattoni grandi come pezzi del lego (le istruzioni semplicissime) ma lo facesse ad una velocità incredibile, costruendo palazzi meravigliosi, ma solo se qualcuno lo guida passo passo (il programma) ... Qualcuno ha infatti definito il computer uno stupido molto veloce !

RAPPRESENTAZIONE INTERNA DELLE INFORMAZIONI

Rimane da sciogliere ancora un nodo importante: come sono rappresentate le informazioni nella RAM e nei registri ? E le istruzioni ? Essenzialmente devono essere trattati numeri e caratteri. Le prime macchine calcolatrici (attenzione: non sto ancora parlando di computer elettronici!) erano grandi grovigli di ingranaggi: pur costruite con una precisione ammirabile, gli attriti ed i tempi di funzionamento di tutte queste parti meccaniche costituivano un limite serio per la velocità di elaborazione.

Mentre un ingranaggio impiega un certo tempo per funzionare, un segnale elettrico va alla velocità della luce (300.000 Km al secondo) ! Come dire, più di sette volte il giro della terra in un secondo ! Sono stati ideati dei dispositivi (circuiti elettronici) in grado sfruttare i segnali elettrici per rappresentare le informazioni ed elaborarle. Si è anche capito che il miglior modo di procedere sarebbe stato quello di rappresentare tutto sotto forma di numeri binari. Cerchiamo di arrivare a questa conclusione per gradi.

I segnali elettrici non sono perfetti: un segnale elettrico, mentre viaggia su un filo, è soggetto a disturbi ed attenuazioni di vario tipo che tendono a distorcerlo: in partenza, cioè, possiede certe caratteristiche (forma, potenza ecc.) ; all'arrivo non sono più esattamente le stesse. Per cui se scelgo due segnali abbastanza simili tra loro e li trasmetto, potrebbe capitare che chi li riceve scambi l'uno per l'altro (errore). Questa possibilità cresce con il numero dei tipi di segnale usati. Questo spiega perché i prototipi di calcolatore che cercavano di usare tanti segnali diversi per rappresentare le cifre numeriche ed i caratteri dell'alfabeto non ebbero molto successo: semplicemente la probabilità di errore era troppo alta ! Certo se i tipi di segnale fossero di meno, diminuirebbe la probabilità di confonderli ... Fin dove possiamo arrivare ? Un solo segnale non basta: sarebbe sempre uguale a se stesso e non porterebbe informazione, ma due ... Due è proprio risultato il numero ottimale: usando solo due segnali il più possibile diversi tra loro, è ben difficile che un segnale venga distorto al punto tale da confonderlo per l'altro. E' nata la logica digitale (binaria), basata cioè sull'uso di due soli simboli (che possiamo far corrispondere alle cifre 0 ed 1). Tutte le informazioni devono essere codificate (rappresentate) con combinazioni diverse di 0 ed 1. Queste cifre, dette binarie, sono anche chiamate bit. Un bit può essere 0 od 1.

I circuiti digitali sono facili da costruire: digitali, così sono chiamati i circuiti in grado di trattare segnali di due tipi, per confrontarli e combinarli. Purtroppo non è possibile neanche accennare alla realizzazione dei circuiti digitali. Vi basti sapere che è relativamente semplice costruire circuiti per memorizzare, sommare, sottrarre, moltiplicare, dividere, confrontare ecc. due numeri binari. Questo perché le regole dell'aritmetica binaria (come si fanno le somme, sottrazioni, moltiplicazioni ecc.) sono addirittura più semplici di quelle che usiamo noi con la nostra aritmetica decimale (le cifre dallo 0 al 9) ! Beh, avete un intero corso di elettronica per toglierv i certe voglie!

Il sistema binario è equivalente a quello decimale: come dire che ogni numero espresso con la nostra notazione decimale è esprimibile anche in binario ed in modo biunivoco. Il termine significa che, dato un qualsiasi numero espresso in decimale, lo si può convertire ottenendo un numero in forma binaria; viceversa se si parte da questo numero binario e lo si converte in decimale, si ritorna al numero di partenza. Questo è molto importante: significa poter continuare per noi a fornire dati in decimale all'elaboratore. Questi saranno convertiti in binario ed elaborati. Il risultato sarà di nuovo, per noi, convertito in binario, con la certezza di non commettere equivoci. Equivalentemente, significa che tutte le operazioni matematiche possibili in decimale lo sono anche in binario e che se una certa operazione in decimale dà un certo risultato, lo stesso verrà fornito in binario dall'operazione corrispondente. Potremmo dire, volgarmente, che il sistema binario è 'potente' quanto quello decimale.

Anche le informazioni non numeriche sono comunque rappresentabili sotto forma di numeri. Consideriamo caso per caso:

Alfanumeriche: le singole cifre numeriche, le lettere dell'alfabeto ed altri caratteri come la punteggiatura: basta far corrispondere ad ogni carattere un ben preciso codice numerico. Uno tra i codici più diffusi è l'ASCII: secondo questo codice, ad esempio la lettera 'A' è codificata con il numero 65, la 'B' con 66 e così via (i numeri precedenti il 65 sono usati per altri caratteri). Naturalmente l'elaboratore saprà quando interpretare un numero come tale o come la rappresentazione di un carattere. Nota: in fondo al libro troverai un appendice con l'intera tabella ASCII.

Grafiche: se il disegno è di tipo tecnico memorizzo le coordinate numeriche di ogni elemento geometrico ed altre caratteristiche (coordinate del centro di una circonferenza e la misura del raggio, le coordinate degli angoli di un rettangolo o degli estremi di una retta ecc.). Se il disegno è pittorico, cioè fatto per punti (pixel), si memorizza per ogni pixel il valore dell'intensità di rosso, verde e blu che lo caratterizza.

Animazioni e filmati: è un caso particolare del precedente. Infatti ogni sequenza di animazione o filmato è formata da tanti fotogrammi, ognuno dei quali può essere considerato un disegno statico, ricadendo nel caso precedente. Un a tecnica molto usata per risparmiare memoria è quella di memorizzare solo alcuni fotogrammi chiave (key frame) e dire per ogni fotogramma intermedio cosa cambia a livello di pixel (compressioni mpeg / divx).

Suoni: ogni suono può essere scomposto, determinando le caratteristiche delle onde sonore che lo costituiscono. Questi caratteristiche sono rappresentate da numeri … Esistono dei formati compressi anche per l'audio (MP3, WMA).

RAPPRESENTAZIONE DELLE ISTRUZIONI:

Si adotta anche in questo caso una rappresentazione numerica. Ecco qui sotto il formato di una generica istruzione:

codice operazione	operando 1 (o suo indirizzo)	operando 2 (o suo indirizzo)	indirizzo risultato

Ogni possibile istruzione è indicata da un codice numerico. Ad esempio la somma potrebbe essere indicata con 001 (ricordiamo che i numeri sono binari, ed usano solo lo 0 e l'1), la differenza con 002 ecc. Dopo aver specificato il tipo di istruzione è necessario indicare gli operandi (ad esempio i numeri da sommare). Qualche volta nell'operazione si mettono direttamente i valori da usare, altre volte si specifica l'indirizzo dove trovare in RAM i valori che servono. Così si potrebbe chiedere di sommare direttamente 0010 (2 in binario) e 0100 (4 in binario) oppure intendere che 0010 è l'indirizzo in RAM in cui trovare il primo numero da sommare e che 0100 è l'indirizzo dove trovare in RAM il secondo numero da sommare. A seconda del codice dell'istruzione la CPU sa come comportarsi. Se l'istruzione produce un risultato è anche possibile indicare a quale indirizzo in RAM depositare il risultato. Ecco come potrebbe apparire un'istruzione completa:

0001	0010	0100	1100

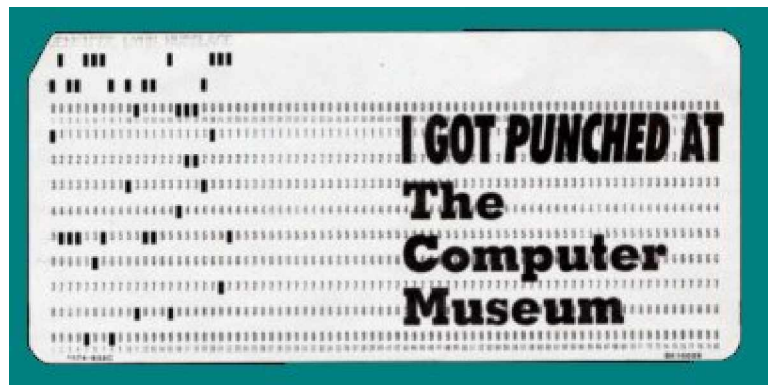
Noterete che sono necessari 16 bit, cioè due byte, per rappresentarla. Infatti, anche se le celle della RAM contengono un byte, spesso sono considerati a gruppi. Il decodificatore riceve proprio la sequenza completa di bit, la scompone in codice operativo, operandi e indirizzo risultato e comunica alla Control Unit le sequenze (microistruzioni) di comandi da attivare. Alcune istruzioni, semplici, occupano un solo byte, altre molti. La CPU non fa in questo caso confusione a causa delle diverse lunghezze: il codice operativo (il primo ad essere letto nella RAM) chiarisce subito di che tipo di istruzione si tratta e di quanti byte necessita, per ogni sua parte. E' per questo che il program counter può essere automaticamente aumentato del giusto numero di byte per 'puntare' alla posizione in RAM in cui si trova la prossima istruzione.

OSSERVAZIONE IMPORTANTE. Sembrerebbe che qualsiasi aspetto della realtà sia perfettamente rappresentabile e trattabile da un elaboratore … ma non è esattamente così ! Dobbiamo convincerci che ciò che viene costruito nelle memorie di un elaboratore è solo un modello (una rappresentazione semplificata, parziale, mai perfetta) della realtà. Certo, quando scriviamo una lettera con Word è esattamente quello che si voleva fare: in questo caso il modello è praticamente perfetto. Ma in altri, per motivi pratici, sono stati imposti dei limiti. Pensiamo ai numeri interi. Più sono grandi e più bit sono necessari per rappresentarli, questo è intuitivo. Per motivi di efficienza e praticità si è deciso di dedicare sempre lo stesso numero di bit per rappresentare un intero: sono usati cioè gli stessi bit per un numero grande che per uno piccolo. E' un po' come scrivere 00128 che è ancora 128: stessa cosa in binario (i bit di più ed inutili sono messi a 0). Già, ma quanti bit ? Se ne vengono usati pochi allora non sarà possibile rappresentare numeri grandi. Se ne vengono usati tanti, tutte le volte che si rappresenta un numero piccolo ne vengono 'sprecati' un certo numero … Qualunque sia la scelta, la conclusione è sempre la stessa: fissato il numero di bit da usare, automaticamente è fissato anche il più grande ed il più piccolo numero intero (positivo e negativo) che si può rappresentare. Stessa cosa per i numeri reali (quelli 'con la virgola'): in questo caso, oltre ai limiti esiste anche un problema di precisione. Infatti un numero può anche essere piccolo ma con un numero infinito di cifre decimali

(pensate al pi greco): se si fissa il numero di bit, più di tante cifre decimali non potranno essere rappresentate. Ecco allora che un numero come 3,145672943456 sarà invece memorizzato come 3,145673, arrotondando la sesta cifra. Ai fini pratici, per la stragrande maggioranza delle applicazioni, la questione è praticamente ininfluente, ma è giusto sapere che questi limiti esistono e che ciò che viene rappresentato all'interno di un sistema di elaborazione può essere soggetto ad approssimazioni. Attenzione: poiché ogni informazione è rappresentata in forma numerica, questo significa che ogni tipo di informazione è soggetto ad approssimazioni (scegliendo, ad esempio, di usare pochi bit per le intensità dei colori di un immagine, la gamma dei colori risultanti sarà povera e discorde da quella reale).

Evoluzione dei linguaggi di programmazione

Ok, tutto questo discorso per avere perfettamente chiaro questo concetto: nella sua forma più primitiva un programma è una sequenza di valori numerici memorizzati in altrettanti byte della RAM; il formato di memorizzazione di questi valori è quello binario. Un programma in questa forma è immediatamente comprensibile al decodificatore e di linguaggio in cui è espresso è chiamato linguaggio macchina. Si parla anche di linguaggio basso livello perché è vicino alla macchina e lontano dal nostro linguaggio naturale.

I primi programmatori non avevano molta scelta: non esisteva la tastiera, non esistevano i supporti di memorizzazione di massa ... Ogni istruzione ed ogni dato veniva comunicato al computer usando le cosiddette schede perforate (ormai reperti archeologici !). Sono tessere di carta, dalle dimensioni di una grossa banconota, su cui i numeri binari sono rappresentati bucando o no piccoli rettangoli disposti in file verticali: un buco all'interno del rettangolino significa 1, niente buco 0. Una fila verticale di rettangolini corrisponde ad un byte: quindi, l'uno di fianco all'altro, sulla stessa scheda possono stare anche più byte (file di rettangolini). Il programmatore, usando un dispositivo chiamato perforatrice, preparava pacchi di schede, corrispondenti ai programmi che voleva far eseguire. Le schede perforate venivano poi lette da un lettore meccanico: questo scandiva ogni scheda rilevando i fori e comunicando 1 o 0 al computer, che a sua volta memorizzava i byte nella RAM.

A parte la scomodità fisica di questo meccanismo (era facilissimo spiegazzare una scheda, con conseguente inceppamento del perforatore o del lettore, oppure invertire la posizione di due schede ecc.), la realizzazione e la manutenzione di un programma non era pane per i denti di tutti:

- bisognava diventare esperti (in materia di programmazione non sono ammesse mezze misure) di quel particolare linguaggio riconosciuto dalla CPU usata, fatto di intricate sequenze di zeri e di uno: provate ad immaginare la fatica di trovare un piccolo errore in una lista di centinaia di migliaia di codici binari ...

- bisognava essere esperti di aritmetica binaria, visto che tutto è espresso in quella forma;

- dato che le istruzioni sono così a basso livello, vicino all'hardware, era necessario conoscere alla perfezione anche quest'ultimo; se cambia l'hardware i programmi dovevano essere pesantemente modificati; se cambiava il microprocessore (la CPU), e con esso l'insieme delle istruzioni accettate, l'intero programma andava riscritto;

Per tutti questi motivi era difficile trovare buoni programmatori e lo sviluppo del software era costoso e lento. Nonostante queste difficoltà i primi abbozzi di sistemi operativi e le prime applicazioni sono stati scritti proprio così ... direttamente linguaggio macchina!

Si parla di linguaggio perché come ogni linguaggio parlato ha un suo vocabolario di 'parole' (l'insieme dei codici delle istruzioni che si possono usare) ed una sua sintassi, cioè le regole con cui si possono formare le 'frasi' (le istruzioni e le sequenze di istruzioni). Ogni frase ha poi il suo 'significato' (l'effetto che produce).

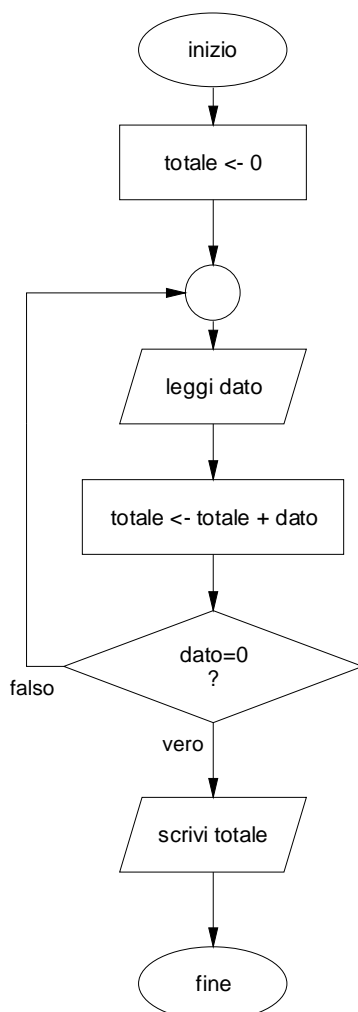
Un primo miglioramento (oltre alla diffusione di dispositivi come le tastiere ed i dischi) è stato introdotto con l'uso di codici mnemonici al posto delle sequenze di 1 e di 0. Il concetto è assai intuitivo: è molto più semplice ricordare

qualche cosa del tipo 'SUM 15 21' che non '100100101011', per indicare che vanno sommati 15 e 21. Il programmatore può ora scrivere il programma usando un linguaggio un po' più lontano da quello macchina ma più vicino al suo modo di esprimersi. Questo linguaggio è chiamato assembly. Naturalmente, prima di sottoporre il programma al computer è necessario un processo di traduzione: i codici mnemonici ed i valori espressi in decimale non sono comprensibile dalla CPU ! Un programma apposito chiamato assemblatore (assembler) , scritto ovviamente in linguaggio macchina, traduce ogni codice mnemonico e valore decimale nella corrispondente sequenza di 1 e di 0. I linguaggi assembly devono essere considerati, come linguaggio macchina, di basso livello.

	BEGIN		;inizio programma
START:	IN	1	;lettura dato da unità 1
	JPZ	FINE	;controllo se il dato letto è 0
	ADD	RIS	;somma il dato a RIS
	STA	RIS	;memorizza la somma in RIS
	JMP	START	;torna a leggere un nuovo dato
FINE:	LDA	RIS	;il ciclo di lettura e somma è finito
	OUT	2	;scrittura del risultato sull'unità 2
	HLT		;fine elaborazione
RIS:	WRD	0	;voce per mantenere somma parziale
	END	START	;fine programma

Qui a lato, un esempio di programma scritto in un linguaggio assembly. Esso calcola la somma di una sequenza di numeri letti (istruzione IN 1) dalla periferica 1 (che potrebbe essere la tastiera) fino a che non viene specificato lo zero come numero da sommare. Quando viene inserito lo zero si termina il ciclo di lettura e si invia il risultato (istruzione OUT 2) alla periferica 2 (che potrebbe essere il video).

NOTA: non sforzatevi di comprendere i dettagli (è ancora troppo presto!)

Beh, certamente meglio di una sfilza di bit, ma ancora piuttosto difficile da leggere! Inoltre questo stesso programma è inservibile con un'altra CPU... Non solo: un programmatore abituato a programmare per un'altra CPU (e quindi con un altro linguaggio assembly) potrebbe non capirci nulla!

Ci vorrebbe una specie di linguaggio universale comprensibile a tutti i programmatori. Uno strumento assai diffuso è rappresentato dai cosiddetti flow chart. La traduzione è diagramma di flusso perché graficamente descrive la sequenza delle istruzioni che verranno eseguite dal computer.

L'idea alla base dei diagrammi di flusso è semplice: invece di usare dei codici letterali che cambiano da una CPU all'altra, usare dei simboli grafici uguali per tutte le CPU. Successivamente il diagramma di flusso guida il programmatore nella scrittura del programma nel linguaggio assembly per la CPU che deve usare. Se un domani gli si chiedesse di riscrivere il programma per un'altra CPU non dovrebbe cominciare da zero ma potrebbe sfruttare lo stesso diagramma di flusso. Addirittura il compito potrebbe essere svolto con la stessa efficienza anche da un altro programmatore.

Qui a lato, il diagramma di flusso che descrive lo stesso algoritmo di somma di una sequenza di numeri terminata da zero vista prima in assembly.

Visivamente è semplice seguire il flusso del programma: si parte dall'ovale con la scritta 'inizio', si giunge ad un riquadro in cui si associa all'identificatore ' totale' il numero zero, ovvio valore di partenza per una somma.

Poi si comanda la lettura di un dato (il pallino che precede indica il punto a cui far ritorno per leggere altri dati); poi si aggiunge al totale il dato letto; nel rombo si decide se è arrivato il momento di terminare o di chiedere altri dati: se il test (confronto del dato letto con il numero zero) dà come risultato ' falso' si segue il percorso indicato dalla freccia con l'etichetta corrispondente e si risale al punto in cui poi, chiedendo un nuovo dato, il ciclo si ripete; se invece il test dà come risultato 'vero' si segue l'altra freccia, il ciclo termina, e si provvede a scrivere il totale calcolato.

Torneremo presto a parlare dei diagrammi di flusso. Infatti questo strumento mantiene la sua validità anche quando introdurremo linguaggi di programmazione più potenti ed espressivi dell'assembly.

I linguaggi assembly sono, infatti, ancora troppo scomodi e soffrono, in fondo, ancora di tutti i problemi elencati per il linguaggio macchina: hanno reso solo un poco più lieve la vita al programmatore, ma sono ancora molto legati alla CPU usata. Il passaggio successivo è stato lo sviluppo di linguaggi ancora più lontani dal linguaggio macchina e, di conseguenza, più vicini al nostro modo di esprimerci. Sono nati i linguaggi ad alto livello. Questi sono caratterizzati da istruzioni molto potenti, con descrizioni facili da ricordare, che corrispondono anche a centinaia di semplici istruzioni in linguaggio macchina o assembly.

```
Program somma;
var totale, dato: integer;

begin
  totale :=0;

  repeat
    read(dato);
    totale := totale + dato
 until dato=0;

write(totale).

end.
```

Ed ecco qui a lato la versione dello stesso algoritmo codificata con il linguaggio di programmazione Pascal (per l'esattezza il suo ‘ dialetto’ Turbo Pascal).

È molto leggibile, compatto. Dopo un'intestazione che assegna un nome al programma, il programmatore dichiara i simboli che vorrà utilizzare indicando anche che sono dei numeri interi (integer).

Il ‘begin’ indica appunto l'inizio del programma. Come prima istruzione si azzera il totale e poi si ripetono (repeat) due istruzioni fino a che (until) non viene introdotto come dato il valore zero: la lettura (read) del dato e la sua aggiunta al totale. Terminato il ciclo si provvede alla scrittura del totale calcolato.

Siamo chiaramente ad un livello più astratto: il programmatore non deve conoscere i meccanismi hardware della stampante o della CPU: ad esempio, deve solo ricordare che il comando di stampa è ‘write’ e che deve specificare tra parentesi ciò che vuole sia stampato. Cambia la stampante ? Il comando rimane sempre questo ! Cambia la CPU ? Il comando rimane sempre questo … Cambia addirittura il tipo di computer? Non devo modificare o riscrivere tutti i programmi: il Pascal rimane Pascal…

Ricordiamo qualche nome di linguaggio ad alto livello tra i più famosi: Fortran (diffusissimo in ambito scientifico e storicamente il primo linguaggio ad alto livello), il ‘C’ (diffusissimo in molti ambiti; la sua capacità di interagire ancora facilmente con l’hardware della macchina gli vale la definizione in effetti di linguaggio di medio livello; molti sistemi operativi, anche per l’efficienza dei programmi che i compilatori ‘C’ sono in grado di produrre, sono per questo motivo in gran parte sviluppati con questo linguaggio), Cobol (diffusissimo in ambito gestionale e commerciale), il mitico Basic (il linguaggio che ha aperto, per la sua semplicità, la porta della programmazione a milioni di utenti anche non esperti di informatica), il Pascal (usatissimo negli ambienti scolastici), il Lisp ed il Prolog (usati per sviluppare programmi di intelligenza artificiale), il C++ (il successore del C). Concludiamo con un linguaggio che sta rapidissimamente conquistandosi i favori dei programmatori perché molto adatto a sviluppare programmi funzionanti su Internet: Java (che molto ha in comune con il C++).

Ad ognuno il suo. Ma se questa è solo una scelta dei principali linguaggi, quanti sono in tutto ? Il numero preciso non è noto, ma è certo che superi qualche centinaio … Come mai ? Per lo stesso motivo per cui esistono in fondo tanti linguaggi anche all’interno del nostro: esiste un linguaggio matematico (osereste negarlo, pensando all’algebra, alle formule matematiche ecc. ?), un linguaggio filosofico, uno giuridico ecc. A seconda dell’ambito applicativo può convenire usare un linguaggio specializzato. Il Fortran deve il suo nome al termine ‘Formula Translator’, cioè traduttore di formule: il suo dizionario è ricco di comandi per calcoli matematici complessi. Il Cobol l’opposto: è ricco però di comandi per gestire efficientemente archivi sui dischi, che è proprio ciò che occorre per sviluppare programmi di gestione aziendale. Attenzione: non è che con un linguaggio si possano fare certe cose ma non altre, solo … più semplicemente. Provate a spiegare le equazioni senza usare termini matematici … E’ possibile, ma che fatica … Per lo stesso motivo sarebbe anche possibile scrivere in Cobol un programma per il calcolo dell’orbita di un satellite, ma che fatica … Certo, alcuni linguaggi sono più specializzati di altri. Alcuni sono invece ad uso generale: il Basic ed il ‘C’ od il C++ non eccellono in nessun campo in particolare ma sono più che adeguati per ogni compito. E’ il mercato che decide la fortuna di un linguaggio.

Ovviamente anche per i linguaggi a medio/alto livello è necessaria una traduzione in linguaggio macchina prima di poterli mandare in esecuzione!

NOTA: tutti i linguaggi diversi dal codice macchina sono chiamati simbolici perché hanno sostituito dei simboli 0.mnemonici alle sequenze di 1 e

Esistono due tipi di traduttori per linguaggi a medio/alto livello:

1. Interpreti. Il programmatore scrive le istruzioni usando un programma di video scrittura (editor). Quando si comanda la partenza del programma, l'interprete legge le istruzioni una alla volta, non opera una vera traduzione ma si limita a riconoscere ciò che le istruzioni vogliono ed a compiere l'azione corrispondente; otteniamo quindi subito, senza attese apprezzabili, il risultato dell'esecuzione di quell'istruzione ma non viene prodotto in modo permanente nessun codice binario: se questa istruzione deve essere eseguita di nuovo, anche per 1000 volte, deve essere reinterpretata.

2 Compilatori: questi sono dei veri traduttori. Come prima, il programmatore scrive le istruzioni usando un programma di video scrittura(editor). Ma, ogni volta che modifica anche un solo carattere del testo del programma e ne richiede l'esecuzione, il programmatore deve comandare ad uno specifico programma (compilatore) la traduzione in linguaggio macchina di tutte le istruzioni che compongono il programma. Come approfondiremo meglio in seguito, il risultato della compilazione prima di poter essere mandato in esecuzione deve essere elaborato anche da un altro programma (linker) che, per così dire, completa l'opera di traduzione iniziata dal compilatore.

Interpreti e compilatori a confronto

Interpreti	Compilatori
Minori tempi di attesa durante lo sviluppo: per vedere il risultato in esecuzione di una modifica ad un'istruzione o della giunta di una nuova istruzione, si deve attendere solo l'interpretazione delle istruzioni che portano a quella che interessa; tutto il resto del programma è come se venisse ignorato. Maggiori tempi totali nel fornire il risultato: l'esecuzione è rallentata dal continuo bisogno di interpretare le righe del programma, anche se sono già state interpretate molte volte. In certe situazioni il tempo di esecuzione di un programma interpretato può essere anche 100 volte superiore a quello del corrispondente programma compilato ... Dipendenza del programma dall'interprete: l'interprete deve sempre essere già pronto e disponibile in memoria. Relativamente semplici da scrivere.	Possibili lunghi tempi di attesa durante lo sviluppo: prima di potere e stare un'istruzione qualsiasi di un programma, è necessario che tutto il programma venga tradotto in linguaggio macchina; se le istruzioni sono poche (qualche centinaio) le potenze elaborative e la velocità degli odierni hard disk (che a volte non vengono addirittura utilizzati perché sono sufficienti le ormai assai capienti e molto più veloci RAM) la velocità della compilazione è tale da sembrare che si stia utilizzando un interprete! Se il progetto consta invece di migliaia, centinaia di migliaia, di milioni di istruzioni (30 milioni per Windows 98!) prima di veder partire il programma in seguito ad una modifica potrebbero rendersi necessari tempi di attesa anche di parecchi minuti... Esecuzione veloce: tempo per la compilazione a parte, il codice binario prodotto dalla compilazione+link può essere eseguito a grande velocità senza bisogno di ulteriori traduzioni. Indipendenza del programma dal compilatore: una volta tradotto in binario il programma, il compilatore non serve più. Più complessi da realizzare rispetto agli interpreti.

Volendo sintetizzare vantaggi e svantaggi, diciamo che con un interprete risulta velocizzata la fase di sviluppo: non ci sono i tempi di compilazione e di link che costringono a ritradurre l'intero programma anche se si sbaglia, letteralmente, una virgola. Un compilatore produce un programma autonomo che garantisce il massimo della velocità in esecuzione (anche decine di volte più veloce): mentre un interprete è perfetto per lo sviluppo del programma, il compilatore serve a generare prodotto finale. Per la verità, queste considerazioni erano molto più vere qualche anno fa: i computer erano molto più lenti nel far funzionare i compilatori (CPU meno potenti e dischi molto più lenti): era allora molto conveniente usare un interprete nelle fasi iniziali (in cui si commettono molti errori) ed il compilatore per i ritocchi finali. Oggi i tempi di compilazione e link sono rapidissimi e l'interprete come strumento è molto meno appetibile.

Sviluppare software, oggi sempre più complicato, è molto costoso. Essere produttivi è indispensabile per avere successo come sviluppatori di software. Gli odierni compilatori e linker sono integrati in ambienti di sviluppo sofisticatissimi, dove alcune parti standard di un programma sono scritte automaticamente ! Ad esempio, mentre un tempo, per gestire una finestra come quelle di Windows, era necessario scrivere decine di istruzioni, oggi il programmatore disegna la finestra con il mouse (come fareste in Paintbrush per un rettangolo) e le istruzioni corrispondenti sono generate in automatico. Lo stesso per molte finestre di dialogo, i bottoni, i menu, le caselle con

gli elenchi, le schede di inserimento dati, i pannelli di ricerca di un file per l'apertura o la registrazione di un archivio ecc. E' sempre disponibile anche un potente strumento per la ricerca degli errori (debugger). Questi sono solo alcuni dei moltissimi bonus che un moderno ambiente per lo sviluppo dei programmi è in grado di offire ...

La catena della programmazione

Il programmatore inizia con lo scrivere le istruzioni (nel linguaggio di programmazione scelto) con un programma chiamato editor. Questo termine inglese significa redattore cioè colui che cura la scrittura di un testo, ed ha quindi un significato abbastanza generico. Molto semplicemente, pensate all'editor come ad un programma di videoscrittura specializzato per la stesura delle istruzioni. Il file scritto con l'editor prende il nome di codice sorgente. (source code). I primi editor erano molto semplici e permettevano solo di scrivere il codice (il testo corrispondente alle istruzioni) e di salvarlo su disco: spettava poi al programmatore mandare in esecuzione, manualmente, il compilatore ed il linker.

Oggi gli editor sono diventati ' intelligenti':

- colorano, o evidenziano automaticamente in altro modo (usando il grassetto, il corsivo, la sottolineatura ecc) le diverse parti di un'istruzione per facilitare la lettura (si parla di sintax highlightining, cioè evidenziazione della sintassi; e così anche decisamente più difficile commettere certi errori (tipo dimenticarsi di chiudere delle parentesi aperte);

- sono in grado di completare automaticamente la scrittura delle istruzioni (code completion): il programmatore inizia a scrivere un comando e l'editor può proporre un elenco di tutti i comandi che iniziano con le stesse lettere; a questo punto programmatore può inserire rapidamente il comando che voleva scrivere senza digitare il resto dei caratteri; può anche usare delle abbreviazioni di pochissime lettere per far inserire un'istruzione composta anche da molte righe che poi verrà completata;

  un'altra caratteristica utilissima è rappresentata dai cosiddetti tooltip (suggerimenti): mentre si sta completando la scrittura di una istruzione complessa sopra il punto di inserimento dei caratteri appare un piccolo riquadro che contiene informazioni sull'istruzione stessa; questo meccanismo consente di risparmiare tantissimo tempo che una volta venire impiegato per andare a consultare un manuale;

- sono integrati con un sistema di aiuto (help in linea) ipertestuale (cioè navigabile facendo clic sui collegamenti che portano ad approfondimenti, argomenti correlati, esempi); a volte l'help è addirittura ipermediale (si possono richiamare filmati, animazioni, commenti vocali ecc.).

- permettono, addirittura, di 'disegnare' parte del programma invece di scriverlo: usando il mouse il programmatore dispone componenti preconfezionati su quella che diventerà la finestra del programma in esecuzione: caselle in cui inserire del testo, bottoni su cui fare clic, menù, tavolozze che appariranno quando si vorrà far scegliere un colore all'utente, finestre di dialogo che permetteranno di cercare un file sul disco quando l'utente comanderà il caricamento di un documento o il suo salvataggio eccetera; per ogni componente scelto l'editor inserirà automaticamente il codice corrispondente! Quando un editor è in grado di funzionare in questo modo è detto visuale.

- sono in grado di richiamare automaticamente il compilatore ed il linker e se uno di questi due programmi trova degli errori mostrare a video i relativi messaggi spostando il punto di inserimento del testo sulla riga del primo errore; se invece il programma non contiene errori può anche essere mandato automaticamente in esecuzione per provarlo: al termine dell'esecuzione si verrà riportati nell'ambiente di lavoro dell'editto. In questo modo il programmatore è in grado di controllare tutti i passaggi senza mai abbandonare le editor. È per queste caratteristiche che oggi, più che di semplici editor, si preferisce parlare di IDE, Integrated Development Envinronment (ambienti integrati di sviluppo).

I file dei codici sorgenti vengono registrati sul disco con un'estensione (la parte del nome del file dopo il punto) che aiuta a riconoscere il linguaggio di programmazione utilizzato. Ecco le principali estensioni: .pas (Pascal), .c (C), .cpp (c ++), .asm (assembly), .asp (ASP), .bas (Basic).

Il compilatore trasforma il codice sorgente nel cosiddetto codice oggetto (object code); ma questo processo di traduzione potrebbe interrompersi per vari motivi:

- viene scoperto un simbolo che non appartiene al cosiddetto dizionario delle parole chiave (keyword) proprie del linguaggio e neppure all'insieme dei simboli aggiunti dal programmatore: ad esempio se il linguaggio prevede un comando write per scrivere qualche cosa sullo schermo, il programmatore potrebbe sbagliare e scrivere wrote; questa parola viene non verrebbe riconosciuta come appartenente al linguaggio; viene allora cercata nell'insieme degli simboli (si parla più correttamente di identificatori) introdotti dal programmatore; quest'ultimo potrebbe aver dichiarato di voler usare un simbolo chiamato area per memorizzare un valore con la virgola in cui far calcolare la superficie del cerchio:

  area := raggio * raggio * 3.14; dai al simbolo area il valore della formula indicata (* significa moltiplicato)

  naturalmente raggio è un altro simbolo introdotto dal programmatore per memorizzare il dato corrispondente alla misura del raggio ...

  se un identificatore non viene trovato neppure nell'insieme definito dal programmatore, allora il compilatore decide che ha trovato un errore lessicale; questa fase viene infatti chiamata analisi lessicale e quella parte del compilatore che la svolge analizzatore lessicale.

- Errori sintattici ed analizzatore sintattico. Anche se viene superata la fase di analisi lessicale ci potrebbero essere ancora errori di tipo sintattico, cioè che violano una regola sintattica di quel linguaggio; le regole sintattiche servono a costruire frasi corrette, al di là del fatto di aver usato singoli termini riconosciuti. Nulla di sorprendente: anche con la lingua italiana dovete rispettare delle regole (soggetto, verbo, complemento oggetto ecc. ricordate?), solo che quelle di un linguaggio di programmazione sono decisamente più semplici! Ed anche in italiano, pur usando tutte parole presenti sul dizionario, è facilissimo scrivere frasi senza senso!

  Facciamo un esempio: molti linguaggi hanno la regola che pretende che ogni istruzione sia terminata con un punto e virgola:

  write('Questa istruzione contiene un errore!')

  Mentre dal punto di vista dell'analizzatore lessicale non ci sono errori, quello sintattico si accorge della mancanza della virgola alla fine dell'istruzione.

Controllo lessicale e sintattico non esauriscono il compito di un compilatore: esiste anche un controllo chiamato semantico che non è però il caso di approfondire in questa sede.

Nota: i codice oggetto hanno di solito come estensione .obj

Il codice oggetto, risultato della compilazione, non è ancora in una forma del tutto eseguibile dalla CPU: alcune parti del codice sorgente non possono essere tradotte dal compilatore perché rappresentano dei riferimenti (link) a dei comandi esterni che non fanno parte direttamente del linguaggio di programmazione in uso ma sono resi disponibili in raccolte chiamate librerie. (library).

Queste ultime sono un preziosissimo strumento per la programmazione: una volta che un problema è stato risolto il codice corrispondente (già tradotto in linguaggio macchina) può essere depositato in una raccolta (la libreria) dalla quale potrà essere estratto per essere incorporato in altri programmi che ne hanno bisogno.

Ad esempio: immaginiamo di avere scritto un comando che disegna sullo schermo una circonferenza note le sue coordinate sullo schermo del centro e la misura del raggio; il codice sorgente viene compilato e tradotto in codice oggetto (da notare che anche in questo caso ci potrebbero essere della parti ancora non tradotte...) ed aggiunto con un programma apposito ad una libreria (immaginiamo che si chiami grafica.lib, dove .lib è naturalmente l'estensione con cui sono di solito individuate su disco le librerie). Decidiamo di chiamare crf (abbreviazione di circonferenza) il comando in questione; le coordinate del centro e la misura del raggio della circonferenza dovranno essere indicate (come in quasi tutti i linguaggi di programmazione) tra parentesi tonde al momento dell'uso del comando stesso,

Immaginiamo ora di scrivere un programma di geometria e di avere la necessità di disegnare alcune circonferenze. Ecco come richiameremmo il comando:

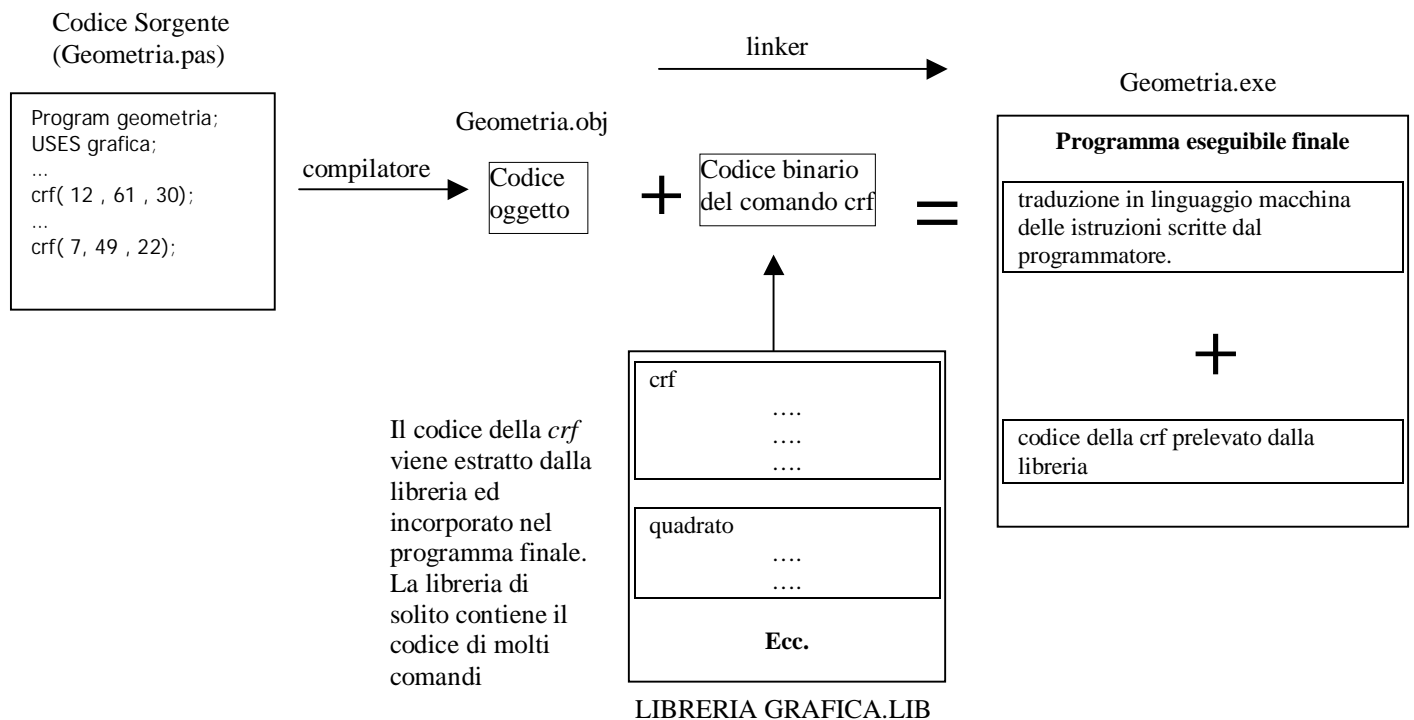
```
programma geometria;

USES grafica;
...
crf( 12 , 61 , 30);
...
crf( 7, 49 , 22);
```

Il programmatore utilizza in questo esempio due volte il comando per disegnare una circonferenza; il centro della prima ha come coordinate 12, 61 ed un raggio pari a 30; similmente per la seconda.

E' molto importante che il programmatore specifichi dove è possibile trovare il codice oggetto che corrisponde al comando crf! Questa informazione viene fornita con 'USES grafica'.

Il compilatore non è in grado di tradurre il comando crf (non appartiene a quelli standard del pascal) ed in mancanza di altre indicazioni dovrebbe interrompersi ed emettere un messaggio di errore (tipo 'simbolo non trovato' in corrispondenza delle righe in cui si richiama il comando crf).

Trovando però l'indicazione di utilizzo della libreria grafica, lascia nel codice oggetto un cosiddetto collegamento (link) non risolto, cioè una specie di segnalibro per ricordare che in quel punto c'è da completare la traduzione.

E' compito del programma linker analizzare tutti i collegamenti irrisolti dei programmi oggetto che gli vengono sottoposti al fine di produrre un unico eseguibile (estensione .exe) e cercare nelle librerie indicate il codice mancante; quest'ultimo verrà estratto in copia ed incorporato nei punti richiesti. Ecco uno schema che esemplifica il meccanismo:

Abbiamo ora tutte le nozioni necessarie per esaminare in dettaglio il procedimento di scrittura di un programma.

Struttura di un programma Pascal

Ogni programma Pascal è costituito da tre sezioni:

Intestazione

Program <nome programma>;

E' una riga che inizia con la parola riservata (keyword) program seguita dal nome che si vuol dare al programma. Le parole riservate sono alcune decine ed il programmatore non può usare identificatori con lo stesso nome di una di esse. Ad esempio, non possiamo chiamare un programma proprio program. Il nome da dare al programma è invece un esempio di identificatore, un nome scelto dal programmatore.

Ci sono alcune regole da rispettare quando scegliamo un identificatore:

- il suo primo carattere non può essere una cifra numerica (il compilatore penserebbe che in quel punto sta iniziando un numero); ad esempio 1arrivato non va bene; arrivato1 invece sì;
- non può contenere spazi; totale iva non va bene; totaleIva o TotaleIva o totale_iva invece sì; notate in particolare l'uso del carattere di sottolineatura _ che consente di 'staccare' le parole come se avessimo inserito uno spazio;
- non può contenere caratteri strani: ad esempio !#?&%$" eccetera; come regola pratica potreste ricordare che possono essere usate solo lettere (maiuscole e minuscole), cifre numeriche ed il carattere di sottolineatura; alcuni linguaggi fanno differenza tra minuscole e maiuscole (e sono detti case sensitive) per cui gli identificatori totale e Totale sono considerati diversi; altri linguaggi, tra cui Pascal, non distinguono invece minuscole maiuscole (e sono detti case insensitive);
- hanno una lunghezza massima, di solito piuttosto ampia (quasi sempre almeno 32 caratteri); attenzione perché è superata la lunghezza massima non è detto che il compilatore segnali errore ma potrebbe semplicemente considerare uguali due identificatori che iniziano con la stessa sequenza di caratteri pari a lunghezza massima; se questa fosse otto, ad esempio, i due identificatori costoTotale e costoTot potrebbero essere considerati lo stesso identificatore, causando situazioni d'errore difficili da scoprire;

Dopo il nome del programma deve essere messo un punto e virgola. Il punto e virgola è molto usato in Pascal (ed in genere da molti linguaggi di programmazione): serve a separare due istruzioni; questo significa che se dopo un'istruzione dev'essere scritta una parola riservata che di per sé non viene considerata un' istruzione il punto e virgola può anche essere omesso. In ogni caso, almeno all'inizio, potreste ricordare come regola semplificata quella di mettere un punto e virgola alla fine di un'istruzione. Ecco allora qualche esempio di intestazione corretta:

```
program prova;        program primoProgramma;    program primo_programma;

program capitolo1;
```

Sezione dichiarativa

Qui il programmatore assegna un nome (identificatore) ai dati che devono essere memorizzati durante l'elaborazione.

Costanti
Se un identificatore viene descritto nella sezione introdotta dalla parola riservata const (abbreviazione di constant, costante) deve anche essere indicato dopo il simbolo dell'uguaglianza (=) il suo valore che non potrà mai variare durante l'esecuzione del programma. Ad esempio il valore del pi greco può essere introdotto in un programma in questo modo:

```
Program geometria;
const
  PI_GRECO = 3.14159265;
```

Nel resto del programma invece di essere costretti a ricordare questa scomoda sequenza di cifre decimali sarà possibile utilizzare la costante PI_GRECO.

Nota: è abitudine dei programmatori usare le maiuscole per gli identificatori che rappresentano una costante.

In un programma possono essere introdotte tutte le costanti che si vogliono. Il tipo del dato (un numero reale, real in Pascal) per una costante viene dedotto dal valore posto dopo il simbolo dell'uguaglianza. Ecco alcuni altri esempi di costanti:

GIORNI_SETTIMANA = 7; esempio di costante di tipo intero (integer in Pascal)

RISPOSTA_AFFERMATIVA = 'S'; esempio di costante di tipo carattere; gli apici sono obbligatori!

MESSAGGIO_ERRORE = 'Sbagliato !!' ; esempio di costante di tipo stringa (sequenza di caratteri racchiusi tra apici, string in Pascal)

Vantaggi dell'uso delle costanti:

- aumentano la leggibilità di un programma: è più facile riconoscere nomi simbolici che numeri complessi;
- aumentano la flessibilità di un programma: se vi è la necessità di cambiare il valore associato ad una costante sarà sufficiente farlo solamente nella sezione const ed automaticamente il nuovo valore verrà utilizzato in ogni punto del programma in cui è stata usata la costante:
- migliorano la sicurezza: il programmatore non potrà neppure per sbaglio cambiare il valore della costante perché il compilatore glielo impedirà emettendo un messaggio di errore

NOTA: la sezione delle costanti può anche non essere presente.

Variabili
Quando invece il valore di un dato può cambiare durante l'esecuzione deve essere memorizzato in una cosiddetta variabile. Le variabili devono essere dichiarate in una sezione, individuata dalla parola chiave var, posta dopo la sezione const:

```
Program geometria;
const
  PI_GRECO = 3.14159265;

var
  raggio : integer;
```

In Pascal le variabili non possono ricevere un valore di partenza direttamente nella sezione dichiarativa ma dovrà essere usato un apposito comando (assegnamento) nella sezione esecutiva. Come certamente immaginate questo stesso comando non può essere invece utilizzato con le costanti. Dopo l'identificatore del nome di una variabile deve essere messo il simbolo ':' (due punti) e poi l'identificatore di un tipo, ed infine il punto e virgola.

Più variabili dello stesso tipo possono essere dichiarate insieme separando gli identificatori con una virgola:

```
var
  base, altezza: integer;
```

Il tipo individua l'insieme dei possibili valori che possono essere assunti da una costante o da una variabile (si parla anche di dominio). Ogni tipo ha infatti i suoi limiti . Ecco la tabella che riassume la situazione per i principali tipi del Pascal:

INTERI

Tipo	***da***	***a***
Byte	0	255
Shortint	-128	127
Integer	-32768	32767
Word	0	65535
Longint	-2147483648	2147483647

REALI

Tipo	***da***	***a***
Real	$2.9x10^{-38}$	$1.7x10^{38}$
Single	$1.5x10^{-45}$	$3.4x10^{38}$
Double	$5.0x10^{-324}$	$1.7x10^{308}$
extended	$1.9x10^{-4951}$	$1.1x10^{4932}$

Esempi di numeri reali: 3.14 4.761E-3;

Il secondo numero è espresso in forma esponenziale: la lettera E seguita da un numero equivale a moltiplicare (o dividere se il numero dopo la E è negativo) per quella potenza del dieci.

Quindi 4.761E-3 = 4.76×10^{-3} = 4.76 / 1000 = 0.00476; 2.45E+4 = 2.45×10^{4} = 24500

NOTA: ci si riferisce a questo tipo di dato anche con la dicitura floating point, a virgola mobile.

TIPO CARATTERE (char). Il tipo char si usa per il singolo carattere (lettera alfabetica, cifra numerica da 0 a 9 o qualsiasi altro simbolo rappresentabile con il P.C. come la punteggiatura, ma anche le lettere dell'alfabeto greco, alcuni caratteri semigrafici tipo cuoricini o faccine, lettere straniere come âåë o simboli matematici). L'elenco completo forma quella che viene chiamata la tabella ASCII (American Standard Code form Information Interchange) che esiste in forma orginale (128 simboli) o estesa (256 simboli). I caratteri vanno racchiusi tra due apici semplici come in 'a', 'A', ']','4'.

Attenzione a non confondere una cifra numerica con il carattere corrispondente: 1 è diverso da '1' ! Anche se in un documento di testo i numeri vengono memorizzati con la sequenza dei codici ascii corrispondenti, questo formato risulta molto inefficiente per svolgere calcoli numerici. Numeri interi e reali vengono rappresentati in modo assai diverso ma questi argomenti verranno affrontati in parallelo nel corso di sistemi.

TIPO BOOLEANO (boolean). I dati boolean possono assumere solo due valori: vero (true) e falso (false).

TIPO STRINGA (string) Un dato string è un insieme di caratteri racchiusi tra apici. La lunghezza massima è 255 caratteri.

Esempi di string: 'Marco', '12/03/1998';

Questi appena visti sono dati semplici o non strutturati: non sono ulteriormente scomponibili. Più avanti nella dispensa verrà introdotta la possibilità di usare dati complessi o strutturati. Vedremo anche come sia possibile per il programmatore aggiungere nuovi dati (user defined type, tipi definiti dall'utente) basandosi su quelli nativi del pascal (standard type, tipi standard).

NOTA: la sezione delle costanti può anche non essere presente.

Procedure e Funzioni
Dopo la sezione delle variabili può essere presente quella dei cosiddetti sottoprogrammi. Si rimanda la discussione di questo argomento ad un momento più appropriato, molto più avanti nella dispensa... Per il momento possiamo comportarci come se questa sezione non esistesse.

Sezione esecutiva (corpo principale del programma)

E' la parte veramente operativa del programma! Corrisponde in pratica alla traduzione dell'algoritmo. La parte dichiarativa è infatti solo preparatoria e serve al compilatore per essere più efficiente (ad esempio se conosce in anticipo i tipi delle variabili può organizzarle più efficientemente nella RAM) e controllare il resto del programma per alcuni tipi di errore (ad esempio tentare di modificare una costante: come potrebbe se non avessimo dichiarato in anticipo quali identificatori considerare associati a valori immutabili?).

Questa sezione inizia con la parola riservata **begin** e termina con la parola riservata **end** seguita da un punto:

TIPI DI ISTRUZIONE CHE SI POSSONO METTERE NELLA SEZIONE ESECUTIVA

assegnamento: memorizza in una variabile una costante oppure il valore di un'altra variabile oppure il risultato di un calcolo (espressione); in Pascal il simbolo dell'assegnamento è questo: **:=**

X := 8; costante   X := Y; variabile   X := Y*2 +4; espressione

```
program EsempiDiAssegnamento;
var
  n1, n2: integer;  x: real;  s: string;
begin
  n1 := 100;  n1 := n2;  n1 := 3*(34 –
n2);
  x := 2;  x := 3.14;  x := n1;   s := 'ciao
come stai?';
end.
```

Il valore che viene assegnato ad una variabile deve essere di tipo compatibile.. Questo non significa per forza identico ma che (almeno) il risultato dell'espressione alla destra del simbolo di assegnamento sia convertibile nel tipo della variabile indicata alla sinistra.

Come regola pratica ricordate questa: un tipo più grande può accoglierne uno più 'piccolo' ma non il viceversa.

```
program AssegnamentiCompatibili;
var
  c: char;  x: real;  s: string;
begin
  x := 3 + 7 (* equivale a 10.0 *);
  c := 'A';
  s := c; (* un carattere 'sta' in una
```

Ad esempio, se X è una variabile reale: x := 7 corrisponde a x := 7.0 e x:=3+7 corrisponde a x := 10.0 (gli interi 7 e l'intero risultato dell'espressione 3+7 possono essere convertiti nel corrispondente numero reale senza problemi). Ma se x fosse una variabile intera, l'assegnamento x := 1.5, ammesso che fosse accettato, dovrebbe trasformare il numero reale 1.5 o nel numero intero uno o nel numero intero due con una perdita di precisione inaccettabile e tale da richiedere un comando esplicito (x := trunc(1.5) ad esempio tronca il numero reale restituendo un intero.

```
stringa *)
end.
```

E’ anche possibile calcolare espressioni facenti uso di parentesi. Un computer non ha però necessità di facilitarsi la lettura di un'espressione utilizzando diversi tipi di parentesi: quadre e graffe non devono essere usate! Bisogna usare invece più livelli di parentesi tonde:

Come siamo abituati noi	Come dev'essere scritto con un linguaggio di programmazione
(5 + 4)	(5 + 4)
(4 -2) * (23 +9)	(4 -2) * (23 +9)
[(4 + 6) / 3] * (23 + 7)	((4 + 6) / 3) * (23 + 7)
{ [(4 + 6) / 3] * (23 + 7) } / (67 – 9)	(((4 + 6) / 3) * (23 + 7)) / (67 – 9)

Principali operatori predefiniti ed operatori relazionali utilizzabili per i principali tipi di dato

CHAR

Ord(carattere)	Codice ascii corrispondente: ord(‘A’) à 65; ord(‘a’) à 97; ord(‘1’) à 49
Chr(codice ascii)	Carattere corrispondente: chr(65) à ‘A’; chr(97) à ‘a’
Upcase(carattere)	Trasforma il carattere in maiuscolo: upcase(‘a’) à ‘A’; upcase(‘A’) à ‘A’

Confronti (predicati): =, < (minore), > (maggiore), <> (diverso), <= (minore od uguale), >= (maggiore od uguale).
NOTA: sono tutti da intendersi in senso alfabetico.

INTERI

+ - *	Somma, sottrazione, moltiplicazione
a DIV b	Quoziente intero tra a e b: 12 div 4 à 3; 14 div 4 à 3;
a MOD b	Resto intero della divisione tra a e b
Succ(numero)	Successore del numero indicato: succ(4) à 5
Pred(numero)	Predecessore del numero indicato: pred(5) à 4
Dec(numero)	Toglie uno a numero: dec(7) à 6
Dec(numero, quantita)	Toglie quantità a numero: dec(12,5) à 7
Inc(numero)	Aggiunge uno a numero: inc(7) à 8
Inc(numero, quantita)	Aggiunge quantità a numero: inc(12,5) à 17
Abs(numero)	Valore assoluto (o modulo) di numero
Sqr(x)	Quadrato
-x	Opposto di x

Possibili errori: **overflow** e **divisione per zero** (division by zero).

Confronti (predicati): =, < (minore), > (maggiore), <> (diverso), <= (minore od uguale), >= (maggiore od uguale).

REALI - ricordare che sono rappresentati nella forma $x = mantissa * 10^{caratteristica}$ ***come in*** $5.28 * 10^6$

+ - * /	Somma, sottrazione, moltiplicazione, divisione
Sqrt (x)	Radice quadrata
Sqr(x)	Quadrato
Sin(x), cos(x), tan(x), arctan(x)	Seno, coseno, tangente, arcotangente di x
Abs(x)	Valore assoluto (o modulo) di x
-x	Opposto di x
Trunc(x)	Tronca la parte frazionare di x
Round(x)	Arrotondamento di x

Possibili errori: **overflow** (caratteristica troppo grande per essere rappresentata), **underflow** (caratteristica troppo piccola per essere rappresentata), divisione per zero, argomento illegale (illegal argument, come in *sqrt(-10)*).

Problemi di precisione: è limitata dal numero di bit dedicati alla mantissa

Problemi di approssimazione: sono dovuti alla conversione tra decimale e binario; ad esempio 0.1 (in decimale) può essere rappresentati in binario solo come numero periodico (0.0001 0001 0001…) che una volta convertito in decimale può dare origine a nome approssimati come 0.0999999 o 9.99999E-2

Confronti (predicati): =, < (minore), > (maggiore), <> (diverso), <= (minore od uguale), >= (maggiore od uguale).

STRINGHE

+	Concatenazione: 'ciao ' + 'a tutti' à 'ciao a tutti'
Copy(s, inizio, quanti)	Restituisce della stringa *s* la sotto stringa che inizia alla posizione *inizio* e continua per *quanti* caratteri

Confronti (predicati): =, < (minore), > (maggiore), <> (diverso), <= (minore od uguale), >= (maggiore od uguale).
NOTA: sono tutti da intendersi in senso alfabetico.

ALTRI COMANDI UTILI

Length(s)	Restituisce alla lunghezza (numero di caratteri) della stringa s
Pos(s1, s2)	Restituisce la posizione della stringa s1 all'interno della stringa s2; se s1 non viene trovata restituisce zer
Val(s, x, errore)	Se possibile trasforma la stringa s nel numero corrispondente e memorizza il risultato nella variabile X; s la conversione riesce nella variabile intera *errore* e ha messo il valore zero; se la conversione non riesce i *errore* viene emesso un valore diverso da zero
Str(x, s)	Converte il numero x nella stringa corrispondente memorizzandola nella variabile stringa s

Incremento/decremento del valore di una variabile.

x := x + 1; x assume il valore attuale incrementato di uno (x diventa x incrementato di uno)

ATTENZIONE: non confondetelo con la scrittura x = y + 1 (test di confronto tra il valore contenuto nella variabile x e quello della variabile y incrementato di uno.

x := x + 5.2; x viene incrementata di 5.2 (evidentemente il tipo di x è *real*)

x := x -10; x viene incrementata di 10

PRINCIPALI OPERATORI IN ORDINE DECRESCENTE DI PRECEDENZA

not	Negazione logica (booleana); esempio: not x=y (espressione vera se il valore della variabile x è diverso da quello di y); Negazione bit a bit (**bitwise**): not 10010 à 01101
*	Moltiplicazione aritmetica
/	Divisione (tra numeri reali)
div	Divisione intera
mod	Modulo (il resto di una divisione intera) 5 mod 2 à 1 18 mod 5 à 3
and	And booleano: (x>4) and (x<12) (espressione vera se x ha un valore compreso tra 4 e 12) And bitwise: 1001 and 1011 à 1101
+	Somma aritmetica, concatenazione di stringhe (‘ciao ‘ + ‘a tutti’ à ‘ciao a tutti’)
-	Sottrazione aritmetica
or	Or booleano: (x<4) or (x>12) (espressione vera se x ha un valore esterno all’intervallo (4,12) Or bitwise: 1001 and 1011 à 1011
=	Test di uguaglianza
<>	Test di non uguaglianza (diverso da); x <> y (espressione vera se il valore della variabile x è diverso da quello di y)
<	Test minore di
>	Test maggiore di
<=	Test minore o uguale di
>=	Test maggiore o uguale di

comandi di ingresso/uscita: servono per acquisire dati dalla tastiera o per visualizzare risultati numerici, messaggi di testo, caratteri semigrafici, primitive geometriche (rette, circonferenze), suoni, stampare ecc.

uscita (scrittura)

write(espressione1, espressione2, …): mostra sullo schermo la sequenza dei valori delle espressioni indicate tra parentesi; *espressione1/2ecc.* può essere praticamente qualsiasi cosa (una costante numerica,una costante stringa, una variabile di tutti i principali tipi, un'espressione che verrebbe prima calcolata ed il cui valore sarebbe quindi visualizzato; nota: non va a capo automaticamente; per cui il successivo comando di visualizzazione su video continuerà sulla stessa linea dello schermo (a meno che si sia già arrivati all'ultima posizione, nel qual caso si va comunque a capo);

writeln(espressione1, espressione2, …): come il comando *write* ed in più si posiziona automaticamente sulla prossima linea dello schermo.

Esempi: write(12); write(‘a’); write(‘ciao mondo’); write(‘Gianni ha ‘, eta, ‘ anni’);
writeln(‘In un secolo ci sono ‘, 60*60*24*365*100.0, ‘secondi’);

NOTA: se non moltiplicassimo per 100.0 (ci avevate badato?) otterremmo un messaggio di errore perché il risultato supererebbe il più grande numero intero (longint) rappresentabile con il turbo Pascal. Indicando invece un numero reale il calcolo viene effettuato nel dominio dei numeri reali.

Il formato del risultato usa la notazione esponenziale: 3.1536000000E+09. E' però possibile indicare degli indicatori di ampiezza: detto in altre parole possiamo scegliere quante posizioni sul video utilizzare per le cifre prima e dopo il punto decimale:

writeln(60*60*24*365*100.0**:10:0**) à 3153600000   :10:0 significa 10 posizioni in tutto, di cui 0 (nessuna) per le cifre dopo il punto decimale

writeln(60*60*24*365*100.0**:13:2**) à 3153600000.00   :13:2 significa 13 posizioni in tutto, di cui 2 per le cifre dopo il punto decimale (bisogna contare anche quest'ultimo)

ingresso (lettura)

readln(variabile): il programma si mette in attesa di un valore che deve essere digitato usando la tastiera; il tipo del dati inserito deve corrispondere a quello della variabile; un quadratino od un trattino lampeggiante indicano lo stato di attesa; il dato si intende digitato completamente quando viene premuto il tasto invio (enter sulle tastiere americane); il valore inserito viene memorizzato nella variabile indicata: da questo momento in avanti diventa disponibile nel programma.

Comandi per il controllo del flusso di esecuzione

Tutti i comandi visti fino ad ora sono del tutto insufficienti per **implementare** (realizzare sul computer) algoritmi anche molto semplici. Immaginiamo ad esempio un programma che chiede a chi lo sta usando di inserire l'età; il programma dovrebbe rispondere con un messaggio appropriato a seconda che ci si trovi di fronte ad un maggiorenne piuttosto che a un minorenne.

Non c'è modo scrivendo semplicemente un'istruzione dopo l'altra di riuscire in questo intento! Per ora, infatti, siamo solo in grado di indicare in sequenza una serie di comandi. Questo è certamente utile ed individua la prima delle cosiddette **strutture fondamentali della programmazione.** Ne mancano due: la **selezione** e l'**iterazione**. La selezione corrisponde la possibilità di far eseguire un blocco di istruzioni piuttosto che un altro a seconda del risultato di un confronto. L'iterazione corrisponde invece la possibilità di far ripetere un blocco di istruzioni fino al verificarsi di una certa condizione.

Due studiosi hanno dimostrato che con sequenza, selezione e l'iterazione possono essere scritti tutti i possibili programmi che un microprocessore è in grado di eseguire. Si parla anche di **programmazione strutturata.**

NOTA: siete caldamente invitati a consultare l'approfondimento sui diagrammi di flusso per vedere come sono rappresentate le strutture fondamentali della programmazione.

Ci sono tre tipi di struttura selettiva: ad una via, a due vie (detta anche binaria), a molte vie.

SELEZIONE AD UNA VIA

Fa eseguire una singola od un blocco di istruzioni (delimitate in questo caso dalla coppia di parole chiave *begin … end*) solo se la condizione indicata e vera (true). Se la condizione è falsa l'istruzione (o il blocco di istruzioni) viene semplicemente ignorato. Ecco la sintassi (a sinistra il caso di singola istruzione ed a destra quello con più di una istruzione:

```
if   condizione  then
   istruzione;
```

ma non sarebbe sbagliato:

```
if condizione then
begin
   istruzioneSingola
end
```

```
if condizione then
begin
  istruzione1;
  istruzione2;
      …
  istruzioneN
end;
```

Dopo *istruzioneN* non è stato dimenticato un punto e virgola: prima di un *end* può essere omessa in quanto il ; serve a separare due istruzioni e l'end non può essere considerato un'istruzione ma un delimitatore.

condizione è un espressione che una volta valutata deve generare un valore booleano (vero/falso); le condizioni si esprimono con gli **operatori relazionali**:

OPERATORE			ESEMPI		
< minore	X < 4	x+y < z	a < b+c	Ord(c) < 67 (1)	Parola1 < Parola2 (2)
<= minore od uguale	X <= 4	x+y < z	a < b+c	Ord(c) <= 67	Parola1 <= Parola2
= uguale	Voto=6	x+y = z	a = b+c	(a+b) div 2 = 4	Cliente1 = Cliente2
>= maggiore od uguale	Totale>=10	x+y >= z	a >= b+c	X >= sqrt(z) (3)	Parola1 >= Parola2
> maggiore	Somma>100	x+y > z	a > b+c	X > sqr(z)	Length(cognome)>12 (4)
<> diverso	x <> y	x+y <> z	a > b+c	X <> sqr(z)	Cognome1 <> Cognome2

(1) *c* è un carattere; la funzione ord restituisce il valore del suo codice ASCII che viene confrontato con il valore 67.
(2) *Parola1* e *Parola2* sono due stringhe; il confronto deve essere inteso in senso lessicografico, cioè considerando la posizione che ciascuna parola occuperebbe sul vocabolario; per cui 'cane' < 'ca in anni libri, e mi sa'; il tutto si applica anche quando la stringa contiene più parole come in 'bel cane' < 'bel gatto'
(3) Il valore della variabile *X* viene confrontato con quello della radice quadrata del valore contenuto nella variabile *z*
(4) Si controlla se la lunghezza in caratteri della stringa *cognome* supera 12

Immaginiamo di voler applicare uno sconto al prezzo di un prodotto solo ai clienti abituali...

```
program ApplicaSconto;
var
   costo: real;    tipoCliente: string;  percentualeSconto: integer;

begin
  writeln('Inserisci costo prodotto: ');   readln( costo );
  writeln('Inserisci eventuale % di sconto: '); readln(percentualeSconto );
  writeln('Inserisci il tipo di cliente: ');   readln( tipoCliente );

 if tipoCliente='abituale' then
    costo := costo – (costo/100) * percentualeSconto;

 writeln('Questo cliente paga : ', costo):
 readln;  (* un readln senza variabile attende fino a che non viene premuto il tasto INVIO)
```

NOTA: in molti degli esempi che verranno proposti non ci si preoccuperà di controllare la correttezza dei dati inseriti tramite la tastiera perché il codice si appesantirebbe e diventerebbe molto più difficile da seguire.

In riferimento al programma qui a lato, ad esempio, l'utente potrebbe inserire un costo pari a zero o negativo, una percentuale di sconto senza senso ed anche una descrizione per il tipo del cliente che potrebbe ' mettere in crisi' il test fatto con l'if: 'Abituale' con la A maiuscola non verrebbe riconosciuto invalido per l'applicazione dello sconto.

end.

Altri esempi.

```
program selezione;
var
  a,b,c: integer;
  s1,s2: string;
  c1,c2: char;
begin
  a:=1; b:=2; c:=1;
  s1:='rossi'; s2:='mario';
  c1:='a';
  c2:='b';

  if a=b then
    writeln('a uguale b');

  if a>3 then
    writeln('a maggiore 3');

  if b<6 then
    writeln('b minore 6');

  if 3>=4 then
    writeln('I n. maggiore uguale II');

  (* il seguente confronto segue l'ordine alfabetico (a<b, b<c, ac>ab,
ecc.) *)
  if c1<=c2  then
    writeln('I car. minore uguale II');

  if s1<>s2 then
    writeln('le stringhe sono diverse');

  if s1>s2 then
    writeln('la I stringa viene dopo la II sul dizionario');

   readln
end.
```

NOTA1: all'interno della parte *then* può essere messa una qualsiasi istruzione Pascal, quindi anche altre strutture di controllo del flusso come un altro 'if … then …'; in pratica ci si ritroverebbe con strutture 'if …then … else' che ne contengono delle altre (vedi più avanti)

NOTA2: dopo un begin (o dopo il *then* se c'è una sola istruzione) abituatevi, andando a capo, a rientrare di un paio di caratteri e ad allineare di conseguenza tutto il blocco delle istruzioni. Dopo l'*end* ritornerete a scrivere le istruzioni allineate con l'*if*. Questa tecnica di scrittura si chiama **indentazione**. Indentare il codice aumenta in modo drastico la leggibilità!

```
Istruzione
If x>2 then
Begin
   Istruzione;
   Istruzione;
   Istruzione
End;
Istruzione;
```

```
Istruzione
If x>2 then
Begin
Istruzione;
Istruzione;
Istruzione
End;
Istruzione;
```

no

ERRORE COMUNE

Il solo fatto di indentare il codice non è sufficiente ad individuare il blocco delle istruzioni dopo il *then*; nell'esempio che segue anche se dall'indentazione è probabile che l'intenzione dal programmatore sia quella di far eseguire le tre istruzioni solo quando il valore della variabile *x* è maggiore di due, in realtà per il compilatore l'istruzione *if … then …* termina dopo il ; di *istruzione1*(le altre due istruzioni sono sempre e comunque eseguite indipendentemente dall'esito del controllo sulla variabile *x).* In pratica il primo spezzone di codice è equivalente a quello centrale; sulla destra, invece, la versione corretta:

```
If x>2 then
   Istruzione1;
   Istruzione2;
   Istruzione3;
```

```
If x>2 then
   Istruzione1;

Istruzione2;
Istruzione3;
```

```
If x>2 then
begin
   Istruzione1;
   Istruzione2;
   Istruzione3
end;
```

SELEZIONE A DUE VIE

Rispetto a quella ad una via viene aggiunta la parte da eseguire quando la condizione falsa:

```
if condizione then
   istruzioneSeVera
else
   istruzioneSeFalsa;
```

```
if condizione then
begin
   Istruzione1SeVera;
   Istruzione2SeVera;
      …
   IstruzioneNSeVera;
end
else
   istruzioneSeFalsa;
```

```
if condizione then
   IstruzioneSeVera
else
begin
   Istruzione1SeFalsa;
   Istruzione2SeFalsa;
      …
   IstruzioneNSeFalsa;
end;
```

```
if condizione then
begin
   Istruzione1SeVera;
   Istruzione2SeVera;
      …
   IstruzioneNSeVera;
end
else
begin
   Istruzione1SeFalsa;
   Istruzione2SeFalsa;
      …
   IstruzioneNSeFalsa;
end;
```

ERRORE COMUNE: mettere il ; prima *dell'else*. In pascal l'istruzione *if … then … else* … è considerata indivisibile e, lo sapete, il ; serve a separare due istruzioni.

Qui sopra sono stati esemplificati tutti i possibili casi che si possono presentare a seconda che sia presente una sola istruzione da eseguire (nella parte *then* o nella parte *else*) piuttosto che un blocco di due o più istruzioni che vanno quindi racchiuse tra un *begin* ed un *end*. Nel primo esempio c'è una sola istruzione da eseguire sia che la condizione sia vera sia che sia falsa. Nel secondo esempio ci sono N istruzioni da eseguire quando la condizione vera ed una sola quando la condizione falsa. Nel terzo esempio c'è una sola istruzione da eseguire quando la condizione vera ed N istruzioni quando la condizione falsa. Nell'ultimo esempio più di un'istruzione da eseguire sia nel caso la condizione sia vera sia nel caso sia falsa.

Se la condizione è vera viene eseguita l'istruzione (o il blocco di istruzioni tra *begin … end*) dopo il *then* e viene invece ignorata la parte *else*. Viceversa, se la condizione è falsa viene ignorata l'istruzione (o il blocco di istruzioni tra *begin … end*) dopo il *then* e viene eseguita la parte *else*.

Condizioni composte

Le condizioni usate fino ad ora sono definite semplici; usando i **connettivi logici** *and*, *or* e *not* possiamo comporne di più complesse (composte).

La condizione composta che usa *and* è vera quando lo sono CONTEMPORANEAMENTE tutte le condizioni coinvolte; se ANCHE UNA SOLA di esse e' falsa, la condizione intera risulterà falsa. Ogni condizione deve essere racchiusa tra parentesi tonde.

```
if (a>=1) and (a<=3) then            (* le parentesi sono OBBLIGATORIE *)
   writeln('a e'' compreso tra 1 e 3'); (* NOTA: per inserire un apostrofo o un accento in una stringa è
                                         necessario raddoppiarlo (altrimenti il compilatore penserebbe
                                         che lì   termina la stringa*)
if (s1='rossi') and (s2='mario') then
  writeln('e'' la persona cercata!!');

if (s1='rossi') and (eta>=18) then
   writeln('rossi e'' maggiorenne, maschio o femmina');

(* le condizioni possono anche essere piu'' di 2 ... devono sempre essere vere
CONTEMPORANEAMENTE *)
if (s1='rossi') and (eta>18) and (s2='femmina') then
   writeln('rossi e'' maggiorenne, femmina');
```

La condizione composta che usa *or* è vera quando lo è ALMENO UNA delle condizioni coinvolte.; è falsa quando le condizioni sono false TUTTE CONTEMPORANEAMENTE.

```
If (mese='Gennaio') or (mese='Marzo') or (mese='Luglio') then
   Numero_giorni := 31;
```

Prestate particolare attenzione al seguente esempio; va letto così: se <bella> oppure <vecchia e ricca> allora .. osservate le parentesi, OBBLIGATORIE: if ( ) or (( ) and ( ))

```
if (s1='bella') or ( (eta>60) and (s2='ricca') ) then
   writeln('la sposo in ogni caso ...');
```

Variante sul tema … (più di una istruzione da eseguire)

```
if (s1='bella') or ( (eta>60) and (s2='ricca') ) then
  begin
  writeln('la sposo in ogni caso ...');
  writeln('poi forse mi pentiro!')
  end
else
  begin
  writeln('in questo caso niente dubbi ...');
  writeln('... meglio scapoli !!')
  end;
```

IF nidificati: gli *if* possono contenere altri *if*; si parla di *if* nidificati:

```
(* troviamo il massimo tra a b c *)
if a>=b then
  if a>=c then
    writeln('il max e'' a')
  else
    writeln('il max e'' c')
else
  if b>=c then
    writeln('il max e'' b')
  else
    writeln('il max e'' c');
```

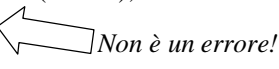
IF in cascata:

```
if condiziome1 then
   …
  else
    if condizione2 then
    …
    else
      if condiziome3 then
      ….
      Else
```

SELEZIONE A MOLTE VIE

Cominciamo con il dire e non è una struttura indispensabile ma solo una comodità. È stata introdotta per sostituire la struttura degli *if* in cascata che rischia di diventare poco leggibile. Ecco qui sotto le due strutture a confronto nel compito di determinare quanti giorni ci sono in un certo mese dell'anno: la variabile *mese* contiene un numero da 1 a 12; la variabile *anno* il numero che corrisponde all'anno di riferimento.

```
If mese=2 then
  If anno mod 4 = 0 then
    Giorni:=29   (* bisestile *)
  Else
    Giorni:=28
Else
  If (mese=1) or (mese=3) or (mese=5)
     or (mese=7) or (mese=8) or (mese=10)
     or (mese=12) then
     Giorni:=31
  Else
    Giorni:=30
```

```
Case eta of
 1..11:   writeln(' sei un poppante!');
 12..15:
    begin
      writeln('OK, dimmi come ti chiami');
       readln( nome );
    end;
 else
   begin
     writeln('Matusa non ammessi …');
     writeln('Premi invio per continuare');
      readln
    end
end;
```

La struttura di destra 'si legge' così: nel caso il valore di mese sia (case mese of …) 2 fai il controllo per il bisestile, nel caso invece il valore sia uno tra 1,3,5,7,8,10,12 i giorni sono 31, altrimenti sono 30. In pratica si elencano uno dopo l'altro i casi possibili (i valori assunti dalla variabile che si sta controllando) e per ciascun caso si indicano le istruzioni da eseguire. La struttura che ne risulta è certamente più lineare e leggibile di quella ottenibile con una serie di *if* ... *then* ... *else* ... annidati e/o in cascata: e più sono i casi da considerare e più conviene usare questa nuova struttura. Ma vediamone in dettaglio la sintassi:

```
Case mese of
 2:  If anno mod 4 = 0 then
        Giorni:=29   (* bisestile *)
      Else
        Giorni:=28;
 1,3,5,7,8,10,12: Giorni:=31;
 else
    Giorni:=30;
end;
```

```
Case eta of
 1..11:   writeln(' sei un poppante!');
 12..15:
    begin
      writeln('OK, dimmi come ti chiami');
       readln( nome );
    end;   <== Non è un errore!
 else
   begin
     writeln('Matusa non ammessi …');
     writeln('Premi invio per continuare');
      readln
    end
end;
```

Dopo la parola riservata *case* deve essere specificata la variabile da controllare: <u>possiamo usare solo variabili di tipo *integer* e *char*</u> (in realtà tutte le variabili di tipo *ordinale*).

Per ogni caso che si presenta possiamo elencare:

- un singolo valore (come il numero 2 che rappresenta febbraio nell'esempio)
- più valori separati da virgola con l'idea che la variabile da controllare può assumere uno qualsiasi di questi valori per ricadere in questo caso
- un intervallo di valori: 1..12 significa da 1a 12, 'a'..'z' significa dalla 'a' alla 'z'
- una situazione mista tra le due precedenti come in 1,3,5-12 che verrebbe interpretato come o il valore 1, o il valore 3, o un valore tra 5 e 12

Dopo l'elenco dei valori che individuano un certo caso si deve mettere simbolo **:** (due punti) e poi o l'unica istruzione da eseguire o il blocco delle istruzioni da eseguire racchiuse tra *begin* e *end.* Ogni caso deve essere separato dal successivo con un punto e virgola.

È possibile, ma non obbligatorio, inserire una parte *else begin* ... *end* con l'idea di indicare le istruzioni da eseguire nel caso il valore della variabile di controllo non trovi riscontro in nessuno dei casi primi elencati. Ad esempio nel controllo del valore della variabile mese se siamo certi che il suo valore è nell'intervallo 1..12, dopo aver controllato che non sia due e tutti i valori che corrispondono a mesi con 31 giorni, la parte *else* non può che corrispondere a casi in cui il mese è uno di quelli con 30 giorni. Naturalmente avremmo anche potuto usare la forma estesa senza la parte *else*:

```
Case mese of
 2:  If anno mod 4 = 0 then
        Giorni:=29   (* bisestile *)
      Else
        Giorni:=28;
 1,3,5,7,8,10,12: Giorni:=31;
 4,6,9,11:   Giorni:=30;
end;
```

Il *case* va sempre concluso con un *end.* Un'ultima particolarità: prima dell'*else* di un *case* <u>deve</u> essere messo un punto e virgola perché altrimenti potrebbe essere confuso con la parte *else* di un *if then else* inserito nell'ultimo caso… Infatti nell'esempio qui sulla destra senza il punto e virgola dopo il 20, *l'else* che dovrebbe essere eseguito quando il soggetto ha più di 18 anni (caso 'pigliatutto' del *case*) verrebbe invece erroneamente associato con l'*if* che controlla le multe.

```
Case eta of
1..13:   paghetta:=5;
14-18:  if multe=0 then
          paghetta:=20;
else
   writeln('Vai a lavorare!')
end;
```

ESERCIZI E RIEPILOGATIVI SULLA SELEZIONE

SEL1. *difficoltà: bassa* Inserita un'età dire se siamo in presenza di un minorenne o maggiorenne.

```
Program maggiorenni1;
uses
  newdelay, crt;
var
  eta: integer;   (* variabile in cui verra' memorizzata l'eta' *)
                (* NOTA: non usare mai le lettere accentate per i
                     nomi delle variabili ! *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Quanti hanni hai? -> ');
  readln(eta);

  if eta>=18 then
    writeln('OK, vedo che sei maggiorenne ...')
  else
    writeln('Sei ancora un poppante, torna tra ', 18 - eta, 'anni !!');

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


(* problemi aperti: cosa succede se chi usa il programma sbaglia ed inserisce
   un eta' negativa o troppo grande per essere vera ?

   modifica il programma per controllare queste possibilita' !!
   trovi la soluzione nel sorgente 'maggior2.pas' *)
end.
```

SEL2. difficoltà: bassa Inserito un numero, dire se e' pari o dispari

```
program paridispari1;
uses newdelay, crt;
var
  numero: integer;   (* variabile in cui verra' memorizzata il numero *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Inserisci un numero e ti diro'' se e'' pari o dispari -> ');
  readln(numero);

  if numero mod 2 = 0 then
    writeln('e'' pari')
  else
    writeln('e'' dispari');

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

SEL3. difficoltà: bassa Inserito un carattere, dire se e' una vocale od una consonante.

```
program consonanteVocale;
uses
  newdelay, crt;
var
  c: char;   (* variabile in cui verra' memorizzata il singolo carattere *)

(* commento: il tipo char può memorizzare un solo carattere; occupa la
   meta' dello spazio di una stringa lunga un solo carattere perché con le
   stringhe un byte supplementare serve a memorizzare la lunghezza della
   stringa stessa; questo non e' invece ovviamente necessario per una
   variabile carattere, la cui lunghezza e'' sempre 1 *)
begin
  clrscr;  (* cancello lo schermo *)

  write('Inserisci un carattere -> ');
  readln(c);


(* PRIMA SOLUZIONE *)
  writeln('PRIMA SOLUZIONE');
  if ( c='a' ) or ( c='e' ) or ( c='i' ) or ( c='o' ) or ( c='u' ) then
    writeln('e'' una vocale')
  else
    writeln('e'' una consonante');

  writeln('--------------------------');
  writeln;
(* commento: questa soluzione non tiene conto delle vocali in maiuscolo *)


(* SECONDA SOLUZIONE *)
  writeln('SECONDA SOLUZIONE');
  if ( c='a' ) or ( c='e' ) or ( c='i' ) or ( c='o' ) or ( c='u' ) or
     ( c='A' ) or ( c='E' ) or ( c='I' ) or ( c='O' ) or ( c='U' )
  then
    writeln('e'' una vocale')
  else
    writeln('e'' una consonante');

  writeln('--------------------------');
  writeln;

(* commento: provate con la A: la prima soluzione non fornisce un risultato
   corretto, la seconda si'; pero'' ci sono molti test ... ecco una soluzione
   piu'' efficiente *)

(* TERZA SOLUZIONE *)
  writeln('TERZA SOLUZIONE');

  (* prima trasformiamo il carattere in maiuscolo e poi facciamo i test
     solo con le maiuscole; upcase e' una funzione che restituisce il
     maiuscolo del carattere fornito come parametro tra parentesi;
     se il carattere era gia' in maiuscolo, rimane in maiuscolo *)
  c := upcase( c );

  if ( c='A' ) or ( c='E' ) or ( c='I' ) or ( c='O' ) or ( c='U' ) then
    writeln('e'' una vocale')
  else
    writeln('e'' una consonante');
```

```
  writeln('--------------------------');
  writeln;


(* QUARTA SOLUZIONE *)
  writeln('QUARTA SOLUZIONE');

  (* soluzione con if in cascata, poco leggibile e lunga *)
  c := upcase( c );

  if ( c<>'A' ) then
    if ( c<>'E' ) then
      if ( c<>'I' ) then
        if ( c<>'O' ) then
          if ( c<>'U' ) then
            writeln('e'' una consonante')
          else
            writeln('e'' una vocale (U)')
        else
          writeln('e'' una vocale (O)')
      else
        writeln('e'' una vocale (I)')
    else
      writeln('e'' una vocale (E)')
  else
    writeln('e'' una vocale (A)');
(* commenti: questa soluzione, piuttosto pesante, consente pero' di
   intraprendere azioni differenziate a seconda delle vocali incontrate;
   i blocchi ELSE sono tutti necessari! diversamente inserendo quella vocale
   non verrebbe stampato nulla! Prova a togliere l'ultimo blocco ed inserisci
   la A ... *)

  writeln('--------------------------');
  writeln;


(* QUINTA SOLUZIONE *)
  writeln('QUINTA SOLUZIONE');

  (* la piu' efficiente, sfruttando una delle funzioni disponibili
     per le stringhe: la funzione pos restituisce la prima posizione di una
     stringa all'interno di un altra; se non la trova restituisce zero;
     ad esempio pos('po','topolino') restituisce 3 perche' la stringa 'po'
     e' stata trovata a partire dalla posizione 3 nella stringa 'topolino' *)

  if pos(c,'aAeEiIoOuU') <> 0 then  (* c trovata da qualche parte *)
    writeln('e'' una vocale')
  else
    writeln('e'' una consonante');

  writeln('--------------------------');
  writeln;

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

SEL4. difficoltà: bassa Inseriti A, B e C dire se B e' compreso tra A e C; in pratica si controlla se B appartiene all'intervallo [A,C]

```
program intervallo;
uses
  newdelay, crt;
var
  a, b, c: real;
  (* variabili in cui verranno memorizzati i numeri *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Inserisci l''estremo sinistro dell''intervallo -> ');
  readln(a);

  write('Inserisci l''estremo destro dell''intervallo -> ');
  readln(c);

  (* se i numeri sono stati inseriti nell'ordine sbagliato, stop *)
  if a>c then
    writeln('Intervallo impossibile.')
  else
  begin
    write('Inserisci un numero e ti diro'' se appartiene all''intervallo -> ');
    readln(b);

    if ( b>=a ) and ( b<=c ) then
      writeln('Appartiene')
    else
      writeln('Non appartiene')

  end;
(* commenti: si e' deciso di comprendere gli estremi (>= e <=) nei controlli;
   notate le parentesi obbligatorie in presenza di piu' di una condizione;
   convincetevi che OR invece di and sarebbe stato un errore: un numero,
   per essere nell'intervallo [a,c] deve essere CONTEMPORANEAMENTE piu'
   grande (al massimo uguale) di a e piu' piccolo (al massimo uguale) di c;

   usando l'OR la condizione sarebbe stata vera anche quando UNA SOLA delle
   condizioni e' vera; ad esempio se l'intervallo fosse [5, 11], ed il numero
   da verificare fosse 13, con  ( b>=a ) or ( b<=c ) andremmo in pratica a
   verificare ( 13>=5 ) or ( 13<=11 ) che risulterebbe vera grazie 13>=5 anche
   se e' falso che 13<=11 (e quindi il punto non appartiene all'intervallo!

   l'or (ma con diverse condizioni) e' il connettivo logico giusto quando
   si vuole controllare che un punto sia ESTERNO ad un intervallo:

    if ( b<a ) or ( b>c ) then
      writeln('Non appartiene')
    else
      writeln('Appartiene')

   infatti un punto NON appartiene se e' piu' piccolo dell'estreo inferiore
   o piu' grande dell'estremo superiore
*)

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

SEL5. difficoltà: bassa Inserite due misure, una in pollici e l'altra in centimetri, dire qual è la maggiore

```
program centimetriPollici;
uses
  newdelay, crt;
var
  cm: real;   (* variabile in cui verra' memorizzate la misura in centimetri *)
  po: real;   (* variabile in cui verra' memorizzate la misura in pollici *)


begin
  clrscr;  (* cancello lo schermo *)

  write('Inserisci la misura in centimetri -> ');
  readln(cm);

  write('Inserisci la misura in pollici -> ');
  readln(po);

  (* ricordando che 1 pollice=2.54 cm ... *)
  if po * 2.54 > cm then
    writeln('La misura in pollici e'' maggiore (', po*2.54:4:2, 'cm)')
  else
    if po * 2.54 < cm then
      writeln('La misura in centimetri e'' maggiore')
    else
      writeln('Le due misure sono equivalenti');


  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

SEL 6. difficoltà: bassa Inserite le misure dei lati di 2 rettangoli dire quale dei due ha la superficie maggiore

```
program confrontaAree;
uses
  newdelay, crt;
var
  b1, b2: real;   (* variabile in cui verranno memorizzate le due basi *)
  h1, h2: real;   (* variabile in cui verranno memorizzate le due altezze *)
  a1, a2: real;   (* variabile in cui verranno memorizzate le due aree *)


begin
  clrscr;  (* cancello lo schermo *)

  write('Inserisci la base del primo rettangolo -> ');

  readln(b1);

  write('Inserisci l''altezza del primo rettangolo -> ');
  readln(h1);

  write('Inserisci la base del secondo rettangolo -> ');
  readln(b2);
```

```
write('Inserisci l''altezza del secondo rettangolo -> ');
  readln(h2);

(* PRIMA SOLUZIONE *)
  writeln('PRIMA SOLUZIONE');

  (* se non interessa memorizzare i valori delle due aree si puo' procedere
     al confronto diretto delle rispettive formule di calcolo *)
  if b1*h1 > b2*h2 then
    writeln('Il primo rettangolo ha la superficie maggiore')
  else
    if b1*h1 < b2*h2 then
      writeln('Il secondo rettangolo ha la superficie maggiore')
    else
      writeln('I due rettangoli hanno la stessa superficie');


  writeln('--------------------------');
  writeln;


(* SECONDA SOLUZIONE (migliore) *)
  writeln('SECONDA SOLUZIONE');

  (* calcoliamo prima le due superfici e salviamo il risultato in a1 e a2 *)
  a1 := b1*h1;
  a2 := b2*h2;

  (* poi usiamo a1 e a2 per i confronti *)
  if a1>a2 then
    writeln('Il primo rettangolo ha la superficie maggiore')
  else
    if a1<a2 then
      writeln('Il secondo rettangolo ha la superficie maggiore')
    else
      writeln('I due rettangoli hanno la stessa superficie');


  writeln('--------------------------');
  writeln;
(* commenti: questa seconda soluzione usa due variabili in più però ha il
   grosso pregio di non far ripetere il calcolo delle aree; inoltre
   permetterebbe di usare i valori delle aree anche in una ipotetica
   continuazione del programma; rende anche più semplice comprendere
   quello che si sta facendo *)

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

SEL7. difficoltà: media Inserita un'età', dire se siamo in presenza di un maggiorenne o di un minorenne; controllare anche eventuali errori di inserimento da parte dell'utente

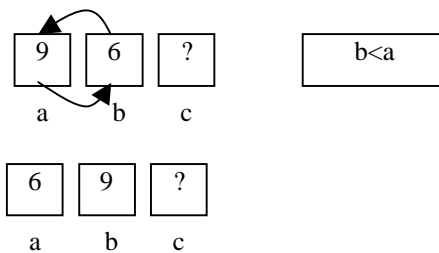
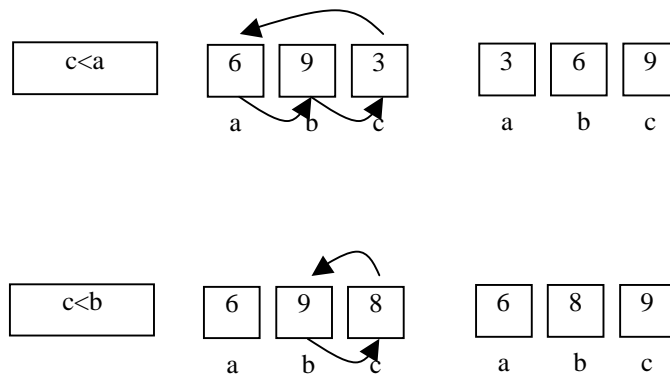
```
program maggiorenni2;
uses  newdelay, crt;
var
  eta: integer;   (* variabile in cui verra' memorizzata l'eta'  NOTA: non usare mai le lettere accentate per i
nomi delle variabili ! *)
begin
  clrscr;  (* cancello lo schermo *)

(* rispetto alla soluzione dell'esercizio maggior.pas indichiamo nella richiesta dell'eta' anche l'intervallo accettato: da
0 anni (neonato) a 120 *)
  write('Quanti hanni hai (0-120) ? -> ');
  readln(eta);

(* SOLUZIONE 1 *)
  (* controlliamo per errori: eta' negative o superiori a 120 *)
  writeln('SOLUZIONE 1');
  if ( eta<0 ) or ( eta>120 ) then
    writeln('ERRORE: l''eta'' deve essere compresa tra 0 e 120 !')
  else
    if eta>=18 then
      writeln('OK, vedo che sei maggiorenne ...')
    else
      writeln('Sei ancora un poppante, torna tra ', 18 - eta, 'anni !!');

(* commenti: notate le parentesi OBBLIGATORIE quando in un if c'e' più di una condizione; notate come sia
possibile iniziare una parte ELSE subito con un altro test; questo tipo di struttura

         if ... then
           ...
         else
           if ... then
             ...
           else
             if ... then
             ecc. ecc.

   e' detta con if in cascata e permette di verificare in successione una sequenza di condizioni

   La soluzione precedente non consente, in caso di errore, di capire se lo sbaglio e' relativo ad un'età' minore di zero
o ad un'età' troppo grande perchè le due condizioni sono testate insieme. Vediamo come ottenere un controllo ancora
piu' accurato rispondendo con messaggi di errore differenziati e piu' accurati*)

(* SOLUZIONE 2 *)
  writeln('------------------------------');
  writeln('SOLUZIONE 2');
  if ( eta<0 ) then
    writeln('ERRORE: l''eta'' non puo''essere negativa!')
  else
    if ( eta>120 ) then
      writeln('ERRORE: l''eta'' non puo''essere maggiore di 120!')
    else
      if eta>=18 then
        writeln('OK, vedo che sei maggiorenne ...')
      else
        writeln('Sei ancora un poppante, torna tra ', 18 - eta, ' anni !!');

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)
end.
```

SEL 8. difficoltà: media Letto un carattere, dire se corrisponde ad una lettera maiuscola e se sì dire se e' una consonante od una vocale.

```
program maiuscola;
var
  c: char;
begin
  write('Inserire un carattere -> ');
  readln(c);

(* PRIMA SOLUZIONE: SENZA CASE *)

(* se il codice ASCII del carattere letto e' compreso tra quelli di A e Z
   allora si tratta di una maiuscola ... *)
writeln('SENZA "CASE" ...');
if ( ord(c)>=ord('A') ) and ( ord(c)<=ord('Z') ) then

begin
  writeln('Hai inserito una maiuscola');

  (* vediamo ora se c e' uguale ad una delle vocali *)
  if (c='A') or (c='E') or (c='I') or  (c='O') or (c='U') then
    writeln('... ed e'' una vocale')
  else
    writeln('... ed e'' una consonante')
end
else
  writeln('Non si tratta di una maiuscola');

(* POI SOLUZIONE CON CASE *)
writeln('CON "CASE" ...');

(* il case piu' esterno ne contiene interamente un altro in un suo caso *)
case ord(c) of
  ord('A')..ord('Z'):  (* .. significa dal valore .. al valore *)
    begin
      writeln('Hai inserito una maiuscola');

      (* questo case e' contenuto nel primo caso di quello piu' esterno *)
      case c of
      'A','E','I','O','U': writeln('... ed e'' una vocale');
      else
        writeln('... ed e'' una consonante')
      end;

    end;
else
  writeln('Non si tratta di una maiuscola');
end;
readln;
end.
```

Ricordo che la funzione ord(c), dove c è un carattere, restituisce il codice ASCII di quest’ultimo. Controllare che ord(c) sia compreso tra ord(‘A’) e ord(‘Z’) significa controllare che il codice ASCII del carattere c sia compreso tra quello della prima e dell’ultima lettera maiuscola e, in definitiva, che c contenga una lettere maiuscola.

SEL 9. difficoltà: media Disegna il diagramma di flusso e scrivi un programma che, letti tre numeri, li metta in ordine crescente

```
program ordina_tre;
var a,b,c,temp: integer;

begin
  write('Inserire il primo numero -> ');
  readln(a);

  write('Inserire il secondo numero -> ');
  readln(b);

  write('Inserire il terzo numero -> ');
  readln(c);

  (* e' necessario assicurarsi che tre numeri siano in ordine *)

  (* prima ordina a e b tra loro *)
  if b<a then
  begin
    temp:=a;
    a:=b;
    b:=temp
  end;

  (* poi metti c al posto giusto ... *)
  if c<a then
  begin
    temp:=a;
    a:=c;
    c:=b;
    b:=temp
  end
  else
    if c<b then
    begin
      temp:=b;
      b:=c;
      c:=temp
    end;

  writeln (a,' ',b,' ',c,' premere INVIO per continuare');
  readln;

end.
```

SEL 10. difficoltà: media Letti tre numeri da tastiera, determinare se possono appartenere ad una progressione aritmetica. TRACCIA: in una progressione aritmetica la differenza tra due numeri consecutivi e' costante. Ad esempio: 1 4 7 10 13 16 ecc. è la progressione in cui ogni numero si ottiene sommando 3 al precedente. La differenza tra due numeri consecutivi qualsiasi è pari a 3.

```
program progressione;

var a,b,c,temp: integer;

begin

 write('Inserire il primo numero -> ');
 readln(a);

 write('Inserire il secondo numero -> ');
 readln(b);

 write('Inserire il terzo numero -> ');
 readln(c);

 (* e' necessario prima assicurarsi che i tre numeri siano in ordine *)

 (* prima ordina a e b tra loro *)
 if b<a then
 begin
  temp:=a;
  a:=b;
  b:=temp
 end;


 (* poi metti c al posto giusto ... *)
 if c<a then
 begin
  temp:=a;
  a:=c;
  c:=b;
  b:=temp
 end
 else
  if c<b then
  begin
   temp:=b;
   b:=c;
   c:=temp
  end;

 if (b-a) = (c-b) then
  writeln('Sono in progressione aritmetica')
 else
  writeln('NON sono in progressione aritmetica');

 writeln ('premere INVIO per continuare');
 readln;
end.
```

SEL 11. difficoltà: alta Verificare se una data inserita da tastiera e' corretta

```
program dataCorretta;
var gg,mm,aa: integer; (* gg=giorno, mm=mese, aa=anno *)
    ok_data: boolean;

begin
  write('Inserire il giorno: ');readln(gg);
  write('Inserire il mese: ');readln(mm);
  write('Inserire l''anno: ');readln(aa);



  (* PRIMA SOLUZIONE SENZA CASE *)
  ok_data:=false; (* se data supera controlli ok_data verra' messo a true *)


  if (gg>0) and (gg<32) and (mm>0) and (mm<13) and (aa>0) then

    if mm=2 then  (* febbraio *)
    begin

      if aa mod 4 = 0 then (* per semplicita': bisestili se multipli di 4 *)
      begin

        if gg<=29 then
          ok_data:=true

      end
      else (* non bisestile *)
        if gg<=28 then
          ok_data:=true

    end

    else (* mese diverso da febbraio ...  *)
      if (mm=1) or (mm=3)or (mm=5) or (mm=7) or
         (mm=8) or (mm=10) or (mm=12)  then (* mese con 31 giorni *)
      begin

        if gg<=31 then
          ok_data:=true

      end
      else (* mese con 30 giorni *)
        if gg<=30 then
          ok_data:=true;


  writeln('Senza case:');
  if ok_data then
    writeln('Data corretta ... ')
  else
    writeln('Data errata !!');

  readln;
```

```
  (* POI SOLUZIONE CON CASE *)
  ok_data:=false;

  if (gg>0) and (gg<32) and (mm>0) and (mm<13) and (aa>0) then

    case mm of
      2: if aa mod 4 = 0 then
         begin

           if gg<=29 then
             ok_data:=true

         end
         else (* non bisestile *)
           if gg<=28 then
             ok_data:=true;

      1,3,5,7,8,10,12: if gg<=31 then
                         ok_data:=true;

    else
      if gg<=30 then
        ok_data:=true;

    end;

  writeln('Con il case:');
  if ok_data then
    writeln('Data corretta ... ')
  else
    writeln('Data errata !!');

  readln;


end.
```

STRUTTURE ITERATIVE

Esamineremo ora i tre tipi classici di struttura iterativa enumerativa (ciclo *for… do*), indefinita con controllo in coda/uscita per vero (ciclo *repeat … until*) e della indefinita con controllo in testa/uscita per falso (ciclo *while*).

Premessa: in realtà potremo scrivere un qualsiasi programma utilizzando solo una delle due forme di struttura iterativa indefinita. Infatti la struttura iterativa enumerativa può essere 'simulata' con una delle altre due, ed è sempre possibile riscrivere un segmento di codice che utilizza una delle due forme di iterazione indefinita usando l'altra. Detto in altre parole tutte le nostre esigenze potrebbero essere soddisfatte o utilizzando solo il ciclo *repeat … until* o usando solo il ciclo *while*.

Come ho avuto già modo di sottolineare per i costrutto *case* l'uso di forme diverse facilita la programmazione in quei casi in cui una delle forme di iterazione esistenti è particolarmente indicata.

Struttura iterativa enumerativa

Si chiama così perché <u>può essere usata solo se il numero di volte che l'istruzione od il blocco di istruzioni deve essere ripetuto è noto nel momento in cui il ciclo inizia</u>, ed è per cui possibile far contare (cioè enumerare) al programma il numero di volte che deve ripetere le istruzioni. Ecco la sintassi:

```
For variabile_di_controllo := valore_inizio to valore_fine do
   Istruzione;
```

Oppure:

```
For variabile_di_controllo := valore_inizio to valore_fine do
begin
   Istruzione1;
   Istruzione2;
    …
   IstruzioneN;
end;
```

La variabile di controllo deve essere di tipo ordinale (*integer, longint o char* anche se quest'ultimo caso è assai raro). Ad essa viene assegnato come valore iniziale quello indicato dopo l'operatore di assegnamento. Se il valore iniziale è inferiore a quello indicato come finale dopo la parola chiave *to* le istruzioni non sono ripetute neppure una volta. Diversamente il ciclo inizia e le istruzioni sono eseguite una prima volta; dopo che è stata eseguita l'ultima istruzione la variabile di controllo viene incrementata di una unità; se il valore è inferiore o uguale a quello indicato come finale il ciclo viene ripetuto. Il ciclo termina quando il valore della variabile di controllo supera quello indicato come valore finale.

L'esempio che segue stampa per cinque volte il messaggio 'ciao!':

```
program cicli;
var i: integer;
begin
  for i:=1 to 5 do
    writeln('ciao!');

   readln;
end.
```

In questo caso il ciclo viene ripetuto una volta

```
program cicli;
var i: integer;
begin
  for i:=1 to 1 do
    writeln('ciao!');

   readln;
end.
```

In questo caso il ciclo non viene ripetuto neanche una volta

```
program cicli;
var i: integer;
begin
  for i:=1 to 0 do
    writeln('ciao!');

   readln;
end.
```

Il valore d'inizio non deve essere per forza 1. Il ciclo che segue stamperà il messaggio sette volte (non dimentichiamo che deve considerare anche il valore 0 nella sequenza che qui riporto per intero -3 -2 -1 0 1 2 3)

```
program cicli;
var i: integer;
begin
  for i:= -3 to 3 do
    writeln('ciao!');

   readln;
end.
```

```
program cicli;
var i, base, altezza, quanti: integer;
begin
  writeln('Quanti triangoli vuoi esaminare?');
  readln( quanti );

  for i:= 1 to quanti do
  begin
    writeln('Inserisci la misura della base del
triangolo');
    readln( base );

    writeln('Inserisci la misura dell'' altezza
corrispondente');
    readln( altezza );

    writeln('La misura della superficie di questo triangolo è: ', base*altezza/2);
     readln;
   end; (* del for *)

 end.
```

Questo esempio chiarisce come non sia necessario conoscere l'esatto valore del numero delle volte che sarà ripetuto il ciclo al momento della scrittura del codice.

È invece necessario che questo valore sia noto al momento in cui il ciclo dovrà essere eseguito: noto non significa sapere quale sarà effettivamente questo valore (è l'utente programma che decide quanti saranno i triangoli esaminati digitando questo un valore quando il programma glielo chiede); significa invece sapere che questo valore è contenuto in una certa variabile (quanti) ed è quest'ultima che può essere utilizzata come valore terminale della variabile di controllo.

Contare come i gamberi: esiste una forma leggermente modificata del costrutto *for* che consente di considerare i valori della variabile di controllo in modo decrescente contando, per così dire, all'indietro. L'esempio seguente stampa sul video i numeri da 10 a 1:

```
program cicli;
var i: integer;
begin
  for i:= 10
downto 1 do
    writeln( i );

   readln;
end.
```

Come vedete, è sufficiente indicare invece di *to* la parola chiave *downto* e come punto di inizio un valore più grande di quello indicato per il punto di fine.

Questo esempio è molto interessante anche per un altro motivo: dimostra come sia **possibile utilizzare nelle istruzioni del ciclo il valore assunto in quel momento dalla variabile di controllo**.

ESERCIZI E RIEPILOGATIVI SUL CICLO FOR

ITE1. difficoltà: bassa stampa dei primi N numeri naturali, con N letto da tastiera

```
program numeri;
uses ewdelay, crt;

var
  i: integer;   (* contatore ciclo *)     n: integer;   (* qui viene memorizzato il numero letto da tastiera *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Fino a che numero devo arrivare? -> ');   readln( n );

  for i := 1 to n do
    writeln( i );

(* commenti: e' ancora materia di disputa tra matematici se considerare lo
   zero un naturale; se lo si vuole considerare tale il codice diventa:

     for i := 0 to n-1 do
       writeln( i );

   infatti se chiedessimo i primi 5 naturali dovremmo ottenere: 0,1,2,3,4
   e sarebbe quindi necessario fermare il ciclo a 4, cioe' n-1

 *)

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)
end.
```

ITE2. difficoltà: bassa stampa dei primi N numeri naturali, con N letto da tastiera; la stampa deve avvenire dal numero piu' grande al piu' piccolo

```
program numeri;
uses  newdelay, crt;

var
  i: integer;   (* contatore ciclo *)    n: integer;   (* qui viene memorizzato il numero letto da tastiera *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Fino a che numero devo arrivare? -> ');   readln( n );

  for i := n downto 1 do   (* NOTATE IL DOWNTO INVECE DI TO !!!! *)
    writeln( i );


  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

ITE3. difficoltà: bassa stampa dei primi N numeri naturali, con N letto da tastiera; a fianco di ciascun numero indicare se e' pari o dispari

```
program numeri;
uses   newdelay, crt;

var
  i: integer;   (* contatore ciclo *)  n: integer;   (* qui viene memorizzato il numero letto da tastiera *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Fino a che numero devo arrivare? -> ');   readln( n );

(* ricordo che l'operatore MOD calcola il resto della divisione tra il numero che viene messo alla sua sinistra e
quello che viene messo alla sua destra; ad esempio 7 mod 3 calcola il resto della divisione tra 7 e 3, quindi 1; 13
mod 8 -> 5; 19 mod 5 -> 4;  quando il primo numero e' multiplo del secondo il resto e' zero: 4 mod 2 -> 0;
15 mod 3 -> 0; 15 mod 5 -> 0; *)

  for i := 1 to n do
  begin
    write( i );

    (* i numeri pari divisi per due danno resto 0 *)
    if i mod 2 = 0 then
      writeln(' numero pari')
    else
      writeln(' numero dispari');

  end;

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)
end.
```

ITE4. difficoltà: bassa Stampa dei numeri dispari minori o uguali a N, con N letto da tastiera

```
program dispari;
uses  newdelay, crt;

var
  i: integer;   (* contatore ciclo *)  n: integer;   (* qui viene memorizzato il numero letto da tastiera *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Fino a che numero devo arrivare? -> ');  readln( n );

  for i := 1 to n do
  begin
    if i mod 2 = 1 then
      writeln( i );

  end;

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)
end.
```

ITE5. difficoltà: bassa Stampa dei primi N numeri naturali (N letto da tastiera) con a fianco i rispettivi quadrati e cubi

```
program potenze;
uses
  newdelay, crt;

var
  i: integer;   (* contatore ciclo *)
  n: integer;   (* qui viene memorizzato il numero letto da tastiera *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Fino a che numero devo arrivare? -> ');
  readln( n );

(* PRIMA SOLUZIONE *)
  writeln('PRIMA SOLUZIONE');

  for i := 1 to n do
    writeln( i, ' ', i*i, ' ', i*i*i );

  writeln('--------------------------');
  writeln;


(* SECONDA SOLUZIONE (sfrutta l'operatore sqr che calcola il quadrato *)
  writeln('SECONDA SOLUZIONE');

  for i := 1 to n do
    writeln( i, ' ', sqr(i), ' ', sqr(i)*i );

  writeln('--------------------------');
  writeln;


(*  TERZA SOLUZIONE
    allinea i numeri usando gotoxy(colonna, riga) per posizionare il cursore alla colonna/riga dello schermo
desiderata; whereY e' una funzione che restituisce il numero di riga dove si  trova il cursore; quindi gotoxy(7,
whereY) significa colonna 7, non muoverti dalla riga su cui ti trovi ...*)
  writeln('TERZA SOLUZIONE');

  for i := 1 to n do
  begin
    write( i );
    gotoxy(7,whereY); write( sqr(i) );
    gotoxy(14,whereY); writeln( sqr(i)*i )
  end;

  writeln('--------------------------');
  writeln;

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)
end.
```

ITE6. difficoltà: bassa Stampa dei numeri interi relativi da -N a +N, con N letto da tastiera

```
program numeri;
uses
  newdelay, crt;

var
  i: integer;   (* contatore ciclo *)
  n: integer;   (* qui viene memorizzato il numero letto da tastiera *)

begin
  clrscr;  (* cancello lo schermo *)

  write('Fino a che numero devo arrivare? -> ');
  readln( n );

  for i := -n to n do
    writeln( i );


  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

ITE7. difficoltà: bassa Stampa dei numeri da INIZIO a FINE, con INIZIO e FINE letti da tastiera. Controllare che INIZIO sia <= FINE *)

```
program numeri;
uses   newdelay, crt;

var
  i: integer;   (* contatore ciclo *)
  inizio, fine: integer;

  temp: integer;  (* variabile di appoggio per scambiare inizio e fine
                  se inseriti nell'ordine sbagliato *)
begin
  clrscr;  (* cancello lo schermo *)

  write('Numero di partenza -> ');
  readln( inizio );

  write('Numero finale -> ');
  readln( fine );

(* PRIMA SOLUZIONE *)
  writeln('PRIMA SOLUZIONE');

  if inizio<=fine then
    for i := inizio to fine do
      writeln( i )
  else
    for i := fine to inizio do
      writeln( i );

  writeln('--------------------------');
  writeln;
(* commenti: se l'utente inserisce inizio e fine nell'ordine sbagliato
   (inizio>fine) si esegue un ciclo con gli estremi invertiti; *)
```

```
(* SECONDA SOLUZIONE *)
  writeln('SECONDA SOLUZIONE');

  (* se inizio e fine sono invertiti, li scambio *)
  if inizio>fine then
  begin
    temp:=inizio;
    inizio:=fine;
    fine:=temp
  end;

  (* ora inizio e fine sono senza dubbio nell'ordine giusto ... *)
  for i := inizio to fine do
    writeln( i );



  writeln('-------------------------');
  writeln;


  writeln('Programma terminato. Premere INVIO per continuare...');

  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

ITE8. difficoltà: bassa generazione di numeri casuali

```
(* random.pas
Con questo esercizio imparerete:

  - l'utilizzo della funzione random per generare numeri in modo casuale
    (utile per simulazioni, giochi ecc.

  - l'utilizzo della funzione randomize per avere sequenze di numeri casuali
    sempre diverse

*)

program prova_random;
uses newdelay crt;

var x,i,num: integer; (* non hanno un significato particolare *)

begin
  clrscr;


(* ESPERIENZA 1 *)
(* random(N) restituisce un numero casuale compreso tra 0 e N-1 *)

  x:=random(5); (* x diventa un numero compreso tra 0 e 4 *)
  writeln('Esperienza 1: ', x);


  (* possiamo usare random direttamente, senza memorizzare il risultato in x *)
  writeln('Esperienza 1: ',random(9));

  readln;
```

```
(* ESPERIENZA 2 *)
(* generiamo una sequenza di 10 numeri compresi tra 0 e 15 *)
  for i:=1 to 10 do
    writeln('Esperienza 2: ', random(16));

  readln;

(* NOTA BENE: tutte le volte che si fa ripartire il programma, random genera
   la stessa sequenza di numeri (osservatelo con l'esperienza n. 2 facendo
   partire il programma piu' volte).

   Se si vuole iniziare ogni volta con una sequenza diversa, usare il comando
   randomize: esso cambia la sequenza basandosi sul tempo scandito
   dall'orologio interno del computer  *)

(* ESPERIENZA 3 *)
(* riscrivete il ciclo dell'esperienza due facendolo precedere dal comando

   randomize; fate ripartire piu' volte il programma e notate come la sequenza
   dell'esperienza 3 cambi tutte le volte *)
  randomize;
  for i:=1 to 4 do

    writeln('Esperienza 3: ', random(16));

  readln;

(* ESPERIENZA 4 *)
(* Invece di un numero possiamo usare come parametro per la random una
   variabile intera *)
  write ('Dimmi un numero: ');
  readln(num);
  writeln('Ora genero una sequenza di 10 numeri tra 0 e ',num - 1);
  for i:=1 to 10 do
    writeln('Esperienza 4: ', random(num));

readln;

(* PROVA DA SOLO ...
   1. Cosa accade indicando 1 come parametro della random ?
   2. Cosa accade indicando 0 come parametro della random ?
   3. Cosa accade indicando un numero negativo come parametro della random ?
   4. Cosa accade indicando un numero reale come parametro della random ?
   5. Cosa accade indicando una variabile integer che contiene un valore
      negativo come parametro della random?
*)

(* ESERCIZI SUGGERITI
ES. 1: trovate un modo per generare numeri tra 1 e N;
------------------------------------------------------------------------
ES. 2: sfruttando quanto scoperto nell'esercizio n. 1, generate una colonna
      di risultati di una schedina del totocalcio;
------------------------------------------------------------------------
ES. 3: generate in modo casuale la sequenza dei nomi per le interrogazioni di una classe con solo quattro alunni che
si chiamano Primo, Secondo, Terzo, e Quarto;

      SUGGERIMENTI: il problema sara' non estrarre piu' volte lo stesso nome; fate corrispondere ogni nome ad un
numero; quando viene estratto quel numero, stampate il nome corrispondente e poi 'cancellate' quel nome
copiando nella variabile che lo memorizza una 'X' al posto del nome; se generate un numero casuale che corrisponde
ad una X (nome gia' estratto)   dovete provare con un altro numero; per sapere quando fermarsi gestite
      un contatore che aumenterete solo quando non troverete una 'X'
------------------------------------------------------------------------ *) end.
```

ITE9. difficoltà: bassa utilizza la random per estrarre numeri da 1 a N

```
program estrai_1_N;
uses newdelay, crt;
var i,n: integer;

begin
  clrscr;

(* dato che random(N) estrae numeri tra 0 ed N-1, e' sufficiente aggiungere
   1 al risultato per avere numeri tra 0+1=1 e N-1+1=N
*)

  write('Che N vuoi usare? -> ');
  readln(n);

  (* estraiamo 20 numeri da 1 a n *)
  for i:=1 to 20 do
    writeln( random(n)+1 );

  (* CONCLUSIONI

     La formula che genera un numero casuale tra 1 e N e':
          RANDOM(N) + 1
  *)
  readln;

end.
```

ITE10. difficoltà: bassa utilizza la random per estrarre numeri tra A e B

```
program estrai_AB;
uses newdelay, crt;
var i,a,b: integer;

begin
  clrscr;

(* random(N) estrae numeri tra 0 ed N-1; immaginiamo di voler estrarre numeri
   tra 5 e 12: e' sufficiente aggiungere a 5 un numero casuale tra 0 e 7
   intuitivamente, infatti, 5+0=5 e 5+7=12; in generale dovremo aggiungere
   al primo estremo (A) un numero casuale tra 0 e B-A
*)

  write('Che A vuoi usare? -> ');
  readln(a);

  write('Che B vuoi usare? -> ');
  readln(b);

  (* estraiamo 20 numeri da A a B *)
  (* NOTA: attenzione! random(B-A) estrarrebbe numeri tra 0 e (B-A) - 1
         e' quindi necessario indicare random (B-A+1) *)
  for i:=1 to 20 do
    writeln( A + random(B-A+1) );

  readln;
end.
```

ITE11. difficoltà: bassa generazione di una colonna di una schedina del totocalcio

```
program colonna_totocalcio;
uses newdelay, crt;
var i: integer;    risultato: integer;
begin
  clrscr;

  randomize;
(* non potendo estrarre la X, useremo un numero al suo posto: il 3; generiamo
   allora 13 numeri da 1 a 3 ... *)
   for i:=1 to 13 do
   begin
     risultato:= random(3)+1;
     if  risultato = 3 then
       writeln('X')
     else
       writeln(risultato)   (* 1 o 2 *)
   end;
   readln;

(* proviamo con una schedina piu' realistica che privilegia gli 1 rispetto alle X e le X rispetto ai 2; si estrae un n. da
1 a 13 e se e' tra 1 e 6 e' come  se fosse uscito l'1, tra 7 e 10 l'X e tra 11 e 13 il 2 *)

   for i:=1 to 13 do
   begin
     risultato:= random(13)+1;
     if  risultato <= 6 then
       writeln(1)
     else
     if risultato<=10 then
       writeln('X')
     else
       writeln(2)
   end;
   readln;
end.
```

ITE12. difficoltà: bassa far scrivere sullo schermo 100 volte la frase 'come programma mi sento un po' stupido ...'

```
program messaggi;
uses  newdelay, crt;
const
  QUANTE_VOLTE=100; (* numero dei messaggi che saranno stampati commento: l'uso di una costante permette
di cambiare molto facilmente il numero dei messaggi senza fare confusione con altri eventuali numeri 100 *)

var
  i: integer;   (* contatore ciclo *)

begin
  clrscr;  (* cancello lo schermo *)

  for i := 1 to QUANTE_VOLTE do
    writeln('come programma mi sento un po'' stupido ...');


  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

ITE13 difficoltà: bassa far scrivere sullo schermo 5 volte la frase 'come programma mi sento un po' stupido ...'; indicare all'inizio di ogni riga il suo numero progressivo (1, 2, 3 ...)

```
program messaggi;
uses
  newdelay, crt;
const

var
  i: integer;   (* contatore ciclo *)

begin
  clrscr;  (* cancello lo schermo *)

(* PRIMA SOLUZIONE *)
  writeln('PRIMA SOLUZIONE');

  for i := 1 to QUANTE_VOLTE do
    writeln('N. riga: ', i ,' come programma mi sento un po'' stupido ...');

  writeln('--------------------------');
  writeln;
```

(* spesso nelle istruzioni di un ciclo for e' utile usare la variabile di controllo del ciclo stesso; in questo esempio, al momento di scrivere ogni riga la variabile di controllo i ha proprio il valore che serve stampare come numero progressivo di riga; infatti quando si stampa per la prima volta il messaggio il suo valore e' uno, quando si stampa il messaggio per la seconda volta il suo valore e' due ecc. *)

```
(* SECONDA SOLUZIONE *)
  writeln('SECONDA SOLUZIONE');

  for i := 1 to QUANTE_VOLTE do
  begin
    write('N. riga: ');(* non va a capo ... *)
    write(i);         (* non va a capo ... *)
    writeln(' come programma mi sento un po'' stupido ...');
  end;

  writeln('--------------------------');
  writeln;
```

(* commenti: la seconda soluzione e' equivalente alla prima; notate come sia INDISPENSABILE aggiungere un begin ed un end per racchiudere ed indicare in modo preciso l'inizio e la fine del blocco di istruzioni che deve essere ripetuto; se infatti avessimo scritto:

```
    for i := 1 to QUANTE_VOLTE do
      write('N. riga: ');
      write(i);
      writeln(' come programma mi sento un po'' stupido ...');
    writeln('--------------------------');
```

il compilatore non potrebbe in alcun modo capire quali sono le istruzioni da ripetere (per quale motivo fermarsi al primo write piuttosto che al secondo?); l'aver allineato i comandi write/writeln serve solo a rendere piu' leggibile il codice per noi ... ma per il compilatore significa comprendere nel ciclo for SOLO LA RIGA SUCCESSIVA:

```
    for i := 1 to QUANTE_VOLTE do
      write('N. riga: ');

    write(i);
    writeln(' come programma mi sento un po'' stupido ...');
    writeln('--------------------------');
```

prova a togliere il begin/end e sperimenta personalmente che il ciclo

```
    stamperebbe per dieci volte la scritta 'N. riga:' e solo una volta il
    resto !!

    A COSA SERVE ALLORA AVER COMPLICATO IL CODICE ??
    RISPOSTA: AD AVER MAGGIOR CONTROLLO, COME MOSTRATO NELLA 'TERZA SOLUZIONE' *)

(* TERZA SOLUZIONE
   questa volta si 'pretende' che il messaggio che precede il numero di riga
   sia scritto in rosso, il numero di riga in giallo e la scritta in verde
*)
  writeln('TERZA SOLUZIONE');

  for i := 1 to QUANTE_VOLTE do
  begin
    textcolor(RED); write('N. riga: ');(* non va a capo ... *)
    textcolor(YELLOW); write(i);        (* non va a capo ... *)
    textcolor(GREEN); writeln(' come programma mi sento un po'' stupido ...');
  end;

  writeln('---------------------------');
  writeln;
(* commenti: solo costruendo il messaggio una porzione alla volta usando il write e' possibile intervallare i comandi
per la selezione del colore *)

  writeln('Programma terminato. Premere INVIO per continuare...');
  readln; (* per dare il tempo di leggere il messaggio *)


end.
```

STRUTTURA ITERATIVA INDEFINITA – il ciclo repeat … until

Serve a ripetere una o più istruzioni fino al momento in cui la condizione specificata dopo la parola chiave *until* diventa vera. Questo momento non è prevedibile da cui la denominazione ‘indefinita’ per questo tipo di ciclo. Ovviamente il ciclo terminerà solo se le istruzioni che vengano ripetute avranno come effetto il raggiungimento della condizione di uscita. Se questo non dovesse avvenire il ciclo si ripeterebbe all'infinito (in gergo si direbbe che il programma è ‘andato in loop infinito’ o, semplicemente,’ andato in loop’)

Nell'esempio che segue si chiede all'utente di inserire un valore maggiore o uguale 10. Se l'utente sbaglia (inserendo un valore minore di 10) riceve un avviso ed il dato viene richiesto. Ecco come viene programmata questa operatività:

```
program prova;
var valore: integer;
begin
   repeat
     writeln(‘Inserire un numero maggiore od uguale 10: ‘);
      readln (valore );

      if valore<10 then
        writeln(‘Errore, riprova …’);
    until valore>=10;
  … resto del programma …
 end.
```

Sarebbe impossibile programmare la stessa operatività utilizzando il ciclo *for*.

Chiaramente è impossibile sapere quante volte sarà necessario ripetere le operazioni a causa di ripetuti errori da parte dell'utente!

Caratteristiche del ciclo repeat:

- il controllo che decide se ripetere o meno il blocco delle istruzioni si trova in fondo al ciclo ed è per questo che il ciclo *repeat* viene detto con controllo in coda; questo comporta che almeno una volta le istruzioni del ciclo vengono eseguite; qualche volta questo è un vantaggio ma altre volte no (quando è necessario controllare prima una condizione senza la quale potrebbe essere addirittura deleterio eseguire le istruzioni); se c'è questa necessità probabilmente è meglio utilizzare l'altro tipo di struttura iterativa indefinita: il ciclo *while* (vedi più avanti)

- il ciclo termina quando la condizione di uscita è vera: per questo motivo si parla anche di ciclo con uscita per vero

Uscita dal ciclo repeat con contatore.

Fondamentalmente si utilizza una variabile per contare quante volte è stato eseguito il ciclo. In pratica possiamo emulare il comportamento di un ciclo *for* ma con tanta flessibilità in più: non siamo ad esempio costretti a contare di uno in uno ma possiamo scegliere di quanto aumentare la variabile di controllo ad ogni esecuzione del ciclo; non siamo neppure costretti ad incrementarla di valori interi: potremmo ad esempio optare per incrementi di millesimi di unità per calcoli di tipo scientifico. Vediamo qualche esempio.

Questo ciclo stampa sul video i numeri da uno a 10. Tutti d'accordo sul fatto che in situazioni di questo tipo sia meglio utilizzare il ciclo *for*: con quest'ultimo non è il programmatore a doversi ricordare che è necessario inizializzare la variabile di controllo (i:=0), incrementarla ad ogni ciclo (i:=i+1) e verificare se è arrivato il momento di interrompere il ciclo, ma è tutto svolto in automatico secondo le indicazioni date dal programmatore sulla prima riga della struttura *for…do*.

```
i:=0;
repeat
  i := i+1;
  writeln( i );
until i=10;
```

Ma ecco una prima applicazione interessante: vengono stampati tutti i multipli di 3 fino al 21. Non è necessario utilizzare l'operatore MOD per testare la divisibilità per tre di ogni singolo valore della variabile di controllo come avremmo fatto con un ciclo *for*: invece di incrementare la variabile di una unità alla volta, si aggiunge tre individuando ogni volta a colpo sicuro un nuovo multiplo!

```
i:=0;
repeat
  i := i+3;
  writeln( i );
until i=21;
```

Uscita dal ciclo repeat con domanda all'operatore.

Da usare in quei casi in cui è necessario far inserire da tastiera una sequenza di dati ed è l'operatore che deve decidere quando sono finiti. Immaginiamo un programma funzionante all'ingresso di una mostra che deve servire a richiedere i dati a grafici di ogni visitatore ed a stampare per ciascuno un cartellino di riconoscimento. Non è ovviamente possibile utilizzare un ciclo *for*: impossibile, infatti, sapere a priori quanti saranno i visitatori… Per lo stesso motivo non è possibile utilizzare un ciclo *repeat* con uscita controllata da un contatore. È invece sufficiente organizzare un ciclo *repeat* in cui l'uscita venga decisa dalla risposta data dall'operatore ad una domanda posta dal computer come ultimo per azione del ciclo:

```
repeat
   writeln('Inserire il nominativo del visitatore: ');
   readln(nome_cognome);

   writeln('Inserire la data di nascita del visitatore: ');
   readln(data_nascita);

   <istruzioni per la stampa del cartellino …>

   writeln('Sono terminati i visitatori?');
   readln( risp );
until risp='S';
```

Si immagina e l'operatore debba rispondere digitando la lettera S.

Uscita dal ciclo repeat con inserimento di un valore convenzionale.

È un miglioramento della tecnica precedente. Quest'ultima, infatti, costringe l'operatore a rispondere anche molte volte alla stessa domanda per dire quando sono terminati i dati. Il 'trucco' consiste nello stabilire che il ciclo deve terminare quando alla richiesta di uno dei dati previsti viene inserito un valore speciale. Riprendendo l'esempio della mostra potremmo decidere che il ciclo termina quando come nominativo del visitatore viene inserita la parola NESSUNO. Ecco come potrebbe apparire il codice:

```
repeat
   writeln('Inserire il nominativo del visitatore (NESSUNO per terminare): ');
   readln(nome_cognome);

   if nome_cognome<>'NESSUNO' then
   begin
      writeln('Inserire la data di nascita del visitatore: ');
      readln(data_nascita);

      <istruzioni per la stampa del cartellino …>

   end;
until nome_cognome='NESSUNO';
```

Da notare che il resto dei dati viene chiesto solo se non è stata inserita la parola convenzionale per terminare il ciclo.

NOTA: la condizione di uscita da un ciclo repeat può essere composta, tipo quelle già viste per la struttura selettiva. L'esempio seguente richiede all'operatore un valore compreso nell'intervallo 1-100:

```
repeat
     writeln('Inserire un valore compreso tra 1 e 100: ');
     readln(valore);
until (valore>=1) and (valore<=100);
```

NOTA: come avete visto dagli esempi, quando c'è più di una istruzione nel ciclo non è necessario utilizzare un blocco *begin … end*: il compilatore infatti è perfettamente in grado di individuare l'inizio e la fine delle istruzioni del ciclo usando le due parole chiave *repeat* e *until* come delimitatori.

STRUTTURA ITERATIVA INDEFINITA – il ciclo while

Si distingue dal *repeat* per queste caratteristiche:

- il controllo che decide la terminazione ciclo è fatto prima delle istruzioni del ciclo stesso: per questo motivo viene anche indicato come ciclo <u>con controllo in testa</u>; se la condizione non è soddisfatta fin dall'inizio il ciclo potrebbe non cominciare neppure, cioè le sue istruzioni non essere eseguite neanche una volta;

- il ciclo termina quando la condizione è falsa: si parla infatti di <u>ciclo con uscita per falso</u>;

Ecco la sintassi:

```
while condizione do
   istruzioneSingola
```

oppure

```
while condizione do
 begin
   istruzione1;
   istruzione1;
       …
   istruzioneN;
 end
```

In generale potremmo dire che le tecniche per decidere la terminazione del ciclo viste per il ciclo *repeat* vanno bene anche per questo ciclo con i dovuti adattamenti resi necessari dalla differente logica:

Stampa dei numeri da 1 a 10. Quando stampa il 10 la condizione diventa falsa (10 non è minore di 10 ma uguale!)

```
i:=0;
while i<10 do
begin
  i := i+1;
  writeln( i );
end;
```

Stampa dei multipli di tre fino al 21.

```
i:=0;
while i<21 do
begin
  i := i+3;
  writeln( i );
end;
```

```
nome_cognome:=''; (* stringa nulla, vuota *)
while nome_cognome<>'NESSUNO' do
begin
  writeln('Inserire il nominativo del visitatore (NESSUNO per
terminare): ');
  readln(nome_cognome);

  if nome_cognome<>'NESSUNO' then
  begin
    writeln('Inserire la data di nascita del visitatore: ');
     readln(data_nascita);

     <istruzioni per la stampa del cartellino …>
```

Da notare la goffaggine di questa soluzione. Poiché è necessario che la variabile di controllo abbia un valore diverso dalla parola 'NESSUNO' per potere iniziare il ciclo, siamo costretti ad usare un assegnamento per dare a questa variabile un valore che soddisfi questo requisito.

È una conferma del fatto che il ciclo *while* sia da preferire solo in quelle situazioni in cui è importante controllare che una certa condizione sia soddisfatta ancor prima di cominciare ad eseguire le istruzioni del ciclo…

```
  end;
end (* del while *)
```

ESERCIZI RIEPILOGATIVI SUI CICLI REPEAT E WHILE

ITE14 difficoltà: media Calcolo delle prime N potenze di 2; deve essere 0<=N<=14

```
(* DOMANDA: PERCHE' NON SI PUÒ SUPERARE L'ESPONENTE 14 ?? *)
program potenzeDue;
var
  n: integer;    (* il numero inserito dall'utente *)

  pot: integer;  (* le potenze di volta in volta calcolate *)
  i: integer;    (* per i cicli *)

begin
  writeln('CALCOLO POTENZE DI 2');
  writeln;

  (* continuo a chiedere un N finche' soddisfa ... *)
  repeat
    writeln('Inserisci n (0<=n<=14): ');
    readln(n);

    if (n<0) or (n>14) then
      writeln('Valore errato! RIPROVA (0<=n<=14)');

  until (0<=n) and (n<=14);


(* VEDIAMO PRIMA LA SOLUZIONE CON IL REPEAT ...*)
  pot:=1; i:=0;

  writeln('SOLUZIONE CON IL REPEAT');
  repeat
    writeln('La potenza ',i, ' del 2 e'': ',pot);
    pot:=pot*2;
    i:=i+1
  until i>n;
  readln;


(* VEDIAMO POI LA SOLUZIONE CON IL WHILE ...*)
  pot:=1; i:=0;

  writeln('SOLUZIONE CON IL WHILE');
  while i<=n do
  begin
    writeln('La potenza ',i, ' del 2 e'': ',pot);
    pot:=pot*2;
    i:=i+1
  end;
  readln;

(* VEDIAMO INFINE LA SOLUZIONE CON IL FOR ...*)
  writeln('SOLUZIONE CON IL FOR');
  pot:=1;
  for i:=0 to n do
  begin
    writeln('La potenza ',i, ' del 2 e'': ',pot);
    pot:=pot*2
  end;
  readln
end.
```

ITE15. difficoltà: bassa Calcolo della media di N numeri inseriti dall'utente

```
program mediaNnumeri;
var
  n: integer;    (* quanti numeri inserire *)
  numero: integer;
  somma: real ;  (* la somma può superare 32767 ... *)
  media: real;
  i: integer;    (* per i cicli *)

begin
  writeln('CALCOLO MEDIA');
  writeln;

  (* continuo a chiedere un N finche' soddisfa ... *)
  repeat
    writeln('quanti numeri vuoi inserire? (N>0): ');
    readln(n);

    if n<=0 then
      writeln('Valore senza senso! RIPROVA (N>0)');

  until n>0;


(* VEDIAMO PRIMA LA SOLUZIONE CON IL REPEAT ...*)
  somma:=0; i:=1;

  writeln('SOLUZIONE CON IL REPEAT');
  repeat

    write('Inserire un numero (',i,') -> ' );
    readln(numero);
    somma:=somma+numero;
    i:=i+1;

  until i>n;

  media:=somma/n;
  writeln('media: ',media:6:2);

  readln;


(* VEDIAMO POI LA SOLUZIONE CON IL WHILE ...*)
  somma:=0; i:=1;

  writeln('SOLUZIONE CON IL WHILE');
  while i<=n do
  begin
    write('Inserire un numero (',i,') -> ' );
    readln(numero);
    somma:=somma+numero;

    i:=i+1;

  end;

  media:=somma/n;
  writeln('media: ',media:6:2);

  readln;
```

```
(* VEDIAMO INFINE LA SOLUZIONE CON IL FOR ...*)
  writeln('SOLUZIONE CON IL FOR');
  somma:=0;

  for i:=1 to n do
  begin
    write('Inserire un numero (',i,') -> ' );
    readln(numero);
    somma:=somma+numero;

  end;

  media:=somma/n;
  writeln('media: ',media:6:2);

  readln;

end.
```

ITE16. difficoltà: media Dato in input un numero intero N, sommare i primi N numeri dispari e verificare che tale somma e' uguale al quadrato di N.

```
program som_disp;
uses newdelay, crt;
var
  i,quanti: integer;  numero,somma: real;

begin
  clrscr;

  (* il controllo che segue*sembra* efficace: provate con 35000;
     e poi con 70000: l'errore non viene rilevato; perche' ?? *)
  repeat
    writeln('Quanti numeri dispari vuoi considerare? (da 0 a ',MAXINT,')');
    readln(quanti);
    if (quanti<0) or (quanti>MAXINT) then
      writeln('Errato, ripetere l''inserimento')
  until (quanti>=0) and (quanti<=MAXINT);

  somma:=0;numero:=1; (* 1 e' il primo n. dispari ...*)
  for i:=1 to quanti do
  begin
    somma:=somma+numero;
    numero:=numero+2; (* individuiamo il n. dispari successivo *)
  end;

  writeln('Considerati i primi ',quanti,' numeri dispari');
  writeln('La loro somma e'': ',somma:10:0);

  (* moltiplico per 1.0 per trasformare in real il risultato
     altrimenti sarebbe integer e limitato a 32767 *)
  writeln('Il quadrato di ',quanti, ' e'' ',quanti*1.0*quanti:10:0);

  if somma=quanti*1.0*quanti then
    writeln('Regola verificata!')
  else
    writeln('Regola non verificata!');

 readln;
end.
```

NOTA: MAXINT è una costante predefinita che rappresenta il più grande numero intero e si può utilizzare; con il turbo Pascal in pratica corrisponde al valore 32.767 ma è meglio usare la costante: perché?

ITE17. difficoltà: media Un giro turistico e' fatto di N tappe, delle quali si introducono da tastiera il nome della citta' di arrivo e i km percorsi. Calcolare il percorso totale e il percorso medio delle tappe

```
program tappe;
uses newdelay, crt;  var
  i,quante_tappe: integer;
  km_tappa,somma_km: real;
  citta: string;

begin

  clrscr;
  repeat
    writeln('Quante sono le tappe? (da 0 a ',MAXINT,')');
    readln(quante_tappe);
    if (quante_tappe<0) or (quante_tappe>MAXINT) then
      writeln('Errato, ripetere l''inserimento')
  until (quante_tappe>=0) and (quante_tappe<=MAXINT);

  somma_km:=0;
  for i:=1 to quante_tappe do
  begin
    writeln('Prossima destinazione ?');
    readln(citta);

    repeat
      writeln('Quanto dista ',citta,' in km?');
      readln(km_tappa);
      if (km_tappa<0) or (km_tappa>MAXINT) then
        writeln('Errato, ripetere l''inserimento')
    until (km_tappa>=0) and (km_tappa<=MAXINT);

    somma_km:=somma_km+km_tappa;
  end;

  writeln('Percorso totale: ',somma_km:5:1,'km');
  writeln('Percorso medio per tappa: ',somma_km/quante_tappe:5:1,'km');
  readln

end.
```

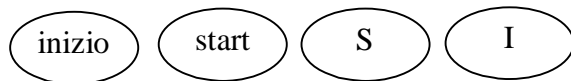
COSA SONO I FLOW CHART

I flow chart sono schemi che descrivono visivamente come procede l'esecuzione di un programma. Essi non sono legati ad uno specifico linguaggio: dato un flow chart, il programmatore può poi usare un qualsiasi linguaggio di programmazione (si tratta, per così dire, di un linguaggio visuale comprensibile a tutti i programmatori). Il flow chart aiuta anche il programmatore a descrivere correttamente un algoritmo (il procedimento risolutivo di un problema).

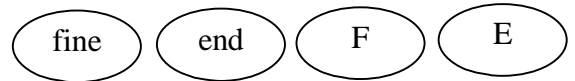
Ogni tipo di istruzione che si può inserire in un programma ha un suo simbolo ed ognuna delle tre strutture fondamentali della programmazione (sequenza, selezione ed iterazione) può essere rappresentata. Esistono anche simboli speciali (inizio programma, fine programma ecc.) che non rappresentano istruzioni vere e proprie ma che sono utili per la costruzione del flow chart.

I PRINCIPALI SIMBOLI

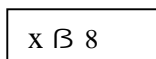
Inizio programma (i simboli sono equivalenti)
puoi ricordare solo un simbolo a scelta ...

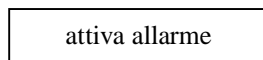
inizio start S I

'S' sta per Start, 'i' sta per Inizio

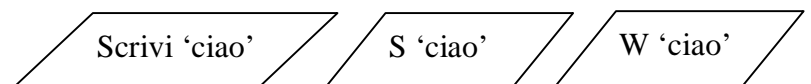
Fine programma (i simboli sono equivalenti)
puoi ricordare solo un simbolo a scelta ...

fine end F E

'F' sta per fine, 'E' sta per End

Assegnamento o altre istruzioni generiche
(esempio: dai ad x il valore 8)

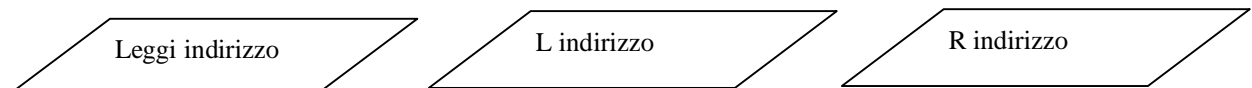
x ß 8

(esempio di istruzione generica)

attiva allarme

Scrittura di un valore sul video (i simboli sono equivalenti)
puoi ricordare solo un simbolo a scelta ...

Scrivi 'ciao' S 'ciao' W 'ciao'

'S' sta per scrivi, 'W' sta per Write

Lettura di un valore scritto con la tastiera e memorizzato nella variabile indicata
(i simboli che seguono sono equivalenti: ricordane uno a tua scelta ..)

Esempio: leggere dove abita una persona e memorizzare l'informazione nella variabile *indirizzo*

Leggi indirizzo L indirizzo R indirizzo

'L' sta per Leggi, 'R' sta per Read

ALTRI ESEMPI

x ß y

dai ad x lo stesso valore di y

x ß y - 1

dai ad x il valore di y diminuito di 1

x ß (5 – 2) * (4 + 3)

dai ad x il valore dell'espressione (5 – 2) * (4 + 3)

S x, y

scrivi sul video prima il valore della variabile x e poi quello della variabile y

L cognome, nome

accetta dalla tastiera un primo valore che memorizzerai nella variabile 'cognome' e poi un secondo valore che memorizzerai nella variabile 'nome'

I simboli visti fino ad ora servono per istruzioni singole. Vediamo ora come si rappresentano le strutture fondamentali della programmazione (sequenza, selezione, iterazione).

Sequenza

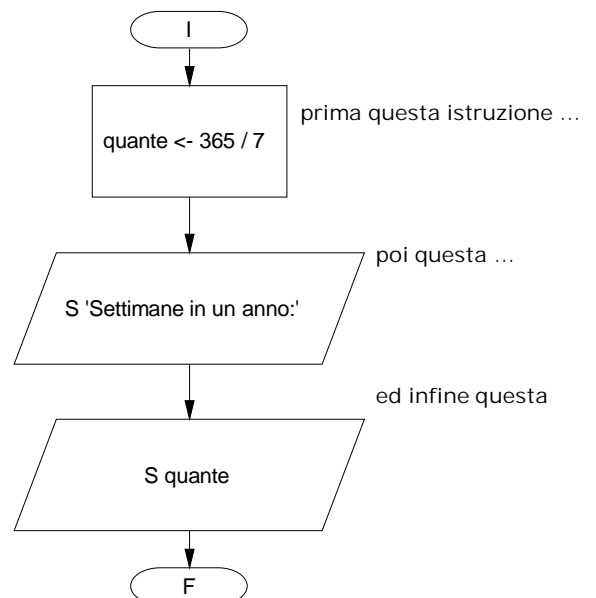
Per indicare che due istruzioni vanno eseguite una dopo l'altra si mettono i loro simboli uno sotto l'altro collegandoli con una freccia. Ecco un esempio.

Calcoliamo quante settimane ci sono in un anno dividendo il numero dei giorni per 7.

Memorizziamo prima il risultato della divisione nella variabile *quante*.

Poi scriviamo un messaggio sul video che 'annuncia' il risultato

Infine scriviamo il risultato, cioè il valore attuale della variabile *quante*.

Il senso della freccia ↓ indica il flusso di esecuzione.

Selezione (decisione, scelta, alternativa)

Il solo fatto di poter indicare un'istruzione dopo l'altra (sequenza) non ci consentirebbe di sviluppare programma interessanti. Ci sono innumerevoli situazioni in cui è necessario fare un 'controllo' ed agire di conseguenza.

Esempi

Se l'anno è bisestile allora considera febbraio con 29 giorni, altrimenti consideralo con 28.

Se l'età è minore di 18 allora applica sconto, altrimenti applica prezzo pieno.

Se la temperatura supera 37 C allora fai suonare l'allarme.

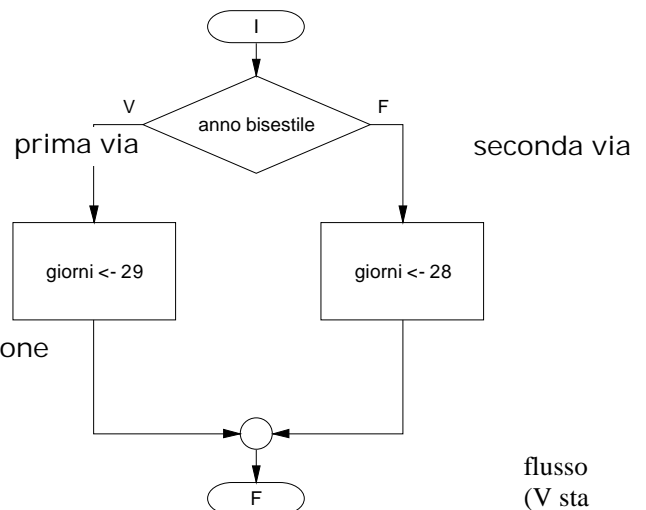
Come avrete osservato, si controlla una *condizione* (anno bisestile ?, età minore di 18 ?, temperatura supera 37 ?) che può risultare vera o falsa. In qualche caso (primi due esempi) ci sono operazioni (diverse) sia nel caso la condizione risulti vera sia nel caso risulti falsa. E' però possibile (terzo esempio) che nel caso la condizione sia falsa non ci sia nulla da fare.

Ecco come si rappresenta in un flow chart la struttura selettiva:

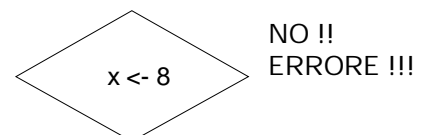
Se l'anno è bisestile allora

considera febbraio con 29 giorni

altrimenti

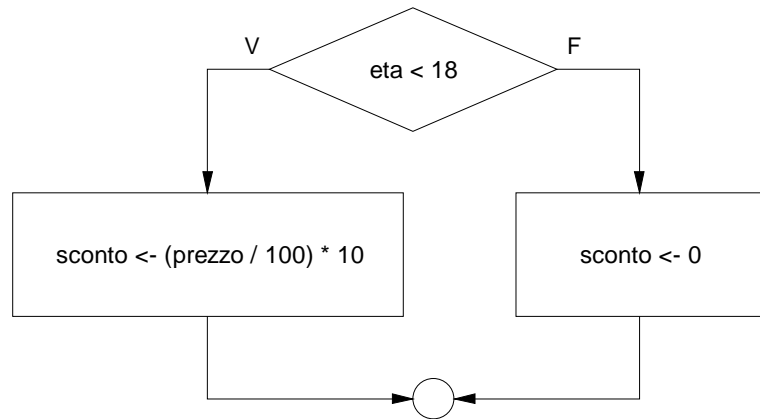
consideralo con 28

La condizione deve essere scritta all'interno del rombo. Il flusso di esecuzione si divide a seconda del risultato del controllo (V sta per Vero e F sta per Falso) e solo la strada 'a destra' o 'a sinistra' viene percorsa.

Le istruzioni nella sezione 'vera' (o 'falsa') possono essere anche molte e non una sola come nel nostro esempio. Quando le istruzioni da eseguire a condizione vera (o falsa) sono terminate il flusso si ricongiunge usando il simbolo chiamato *connettore*.

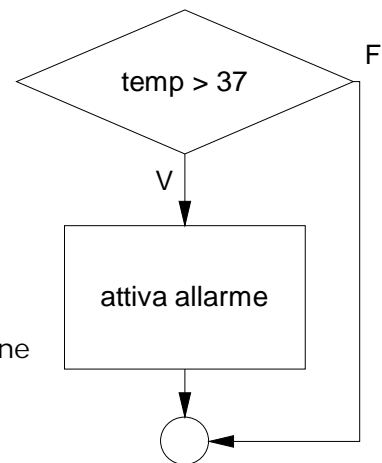
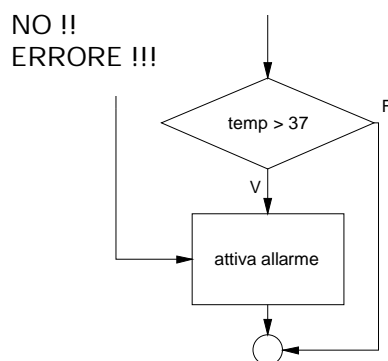
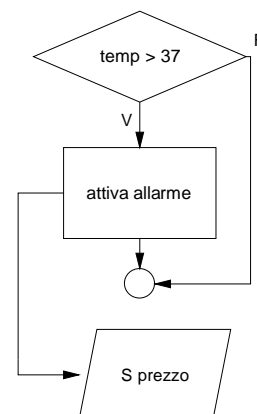
Attenzione: nel rombo potete mettere solo condizioni, non
Sarebbe quindi un errore grave il seguente:

Se l'età è minore di 18 allora

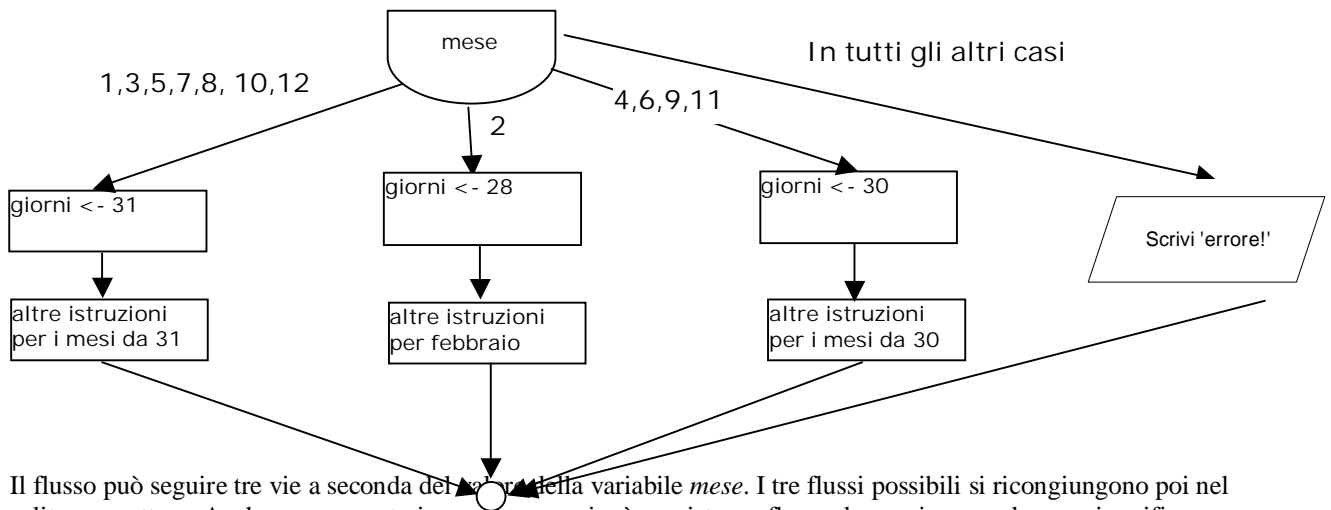
applica sconto

altrimenti

applica prezzo pieno.

Nota: nel flow chart immaginiamo che le variabili *eta* e *prezzo* abbiano già un valore valido e che i calcoli indicati siano quindi fattibili. Lo sconto (10%) viene calcolato dividendo il prezzo pieno per 100 (calcolando così l'1%) e moltiplicando il risultato per 10 (ottenendo il 10%).

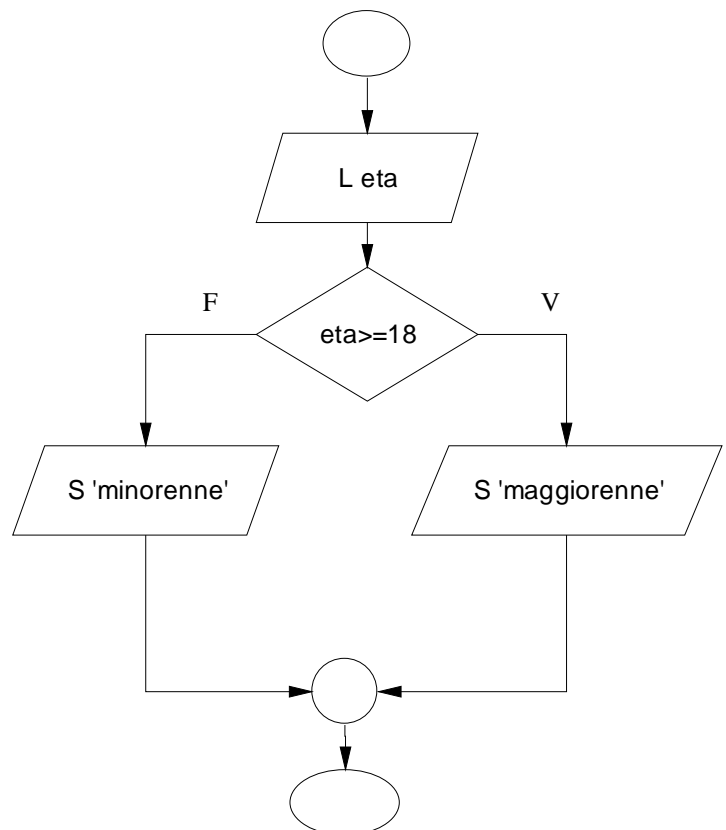
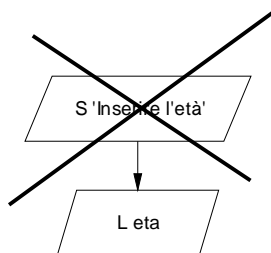
Se la temperatura supera 37 C allora
fai suonare l'allarme.

Come potete vedere se non c'è nulla che deve essere fatto quando la condizione è falsa, si congiunge la parte 'falso' direttamente con il connettore.

NOTA IMPORTANTE: in ogni diagramma che rappresenta la selezione ci deve essere sempre un solo punto di ingresso (la freccia entrante nel rombo) ed un solo punto di uscita (il connettore finale). Evitate quindi frecce che dall'interno escono verso altri punti del diagramma o che dall'esterno giungono in un punto interno:

Selezione a molte vie

Esiste anche un'utile variante che prevede molte vie a seconda del valore di una variabile. Immaginiamo che la variabile *mese* contenga un valore da 1 a 12 che rappresenta uno dei mesi dell'anno. Di nuovo, vorremmo sapere quanti giorni considerare. Vi ricordo che i mesi con 31 giorni sono 1 (gennaio), 3 (marzo), 5 (ecc.), 7, 8, 10, 12; quelli con 30 giorni sono 4 (aprile), 6, 9, 11; immaginiamo per semplicità che febbraio (2) ne abbia sempre 28.

Il flusso può seguire tre vie a seconda del valore della variabile *mese*. I tre flussi possibili si ricongiungono poi nel solito connettore. Anche se non usato in questo esempio, è previsto un flusso da seguire quando non si verifica nessuno dei casi precedenti.

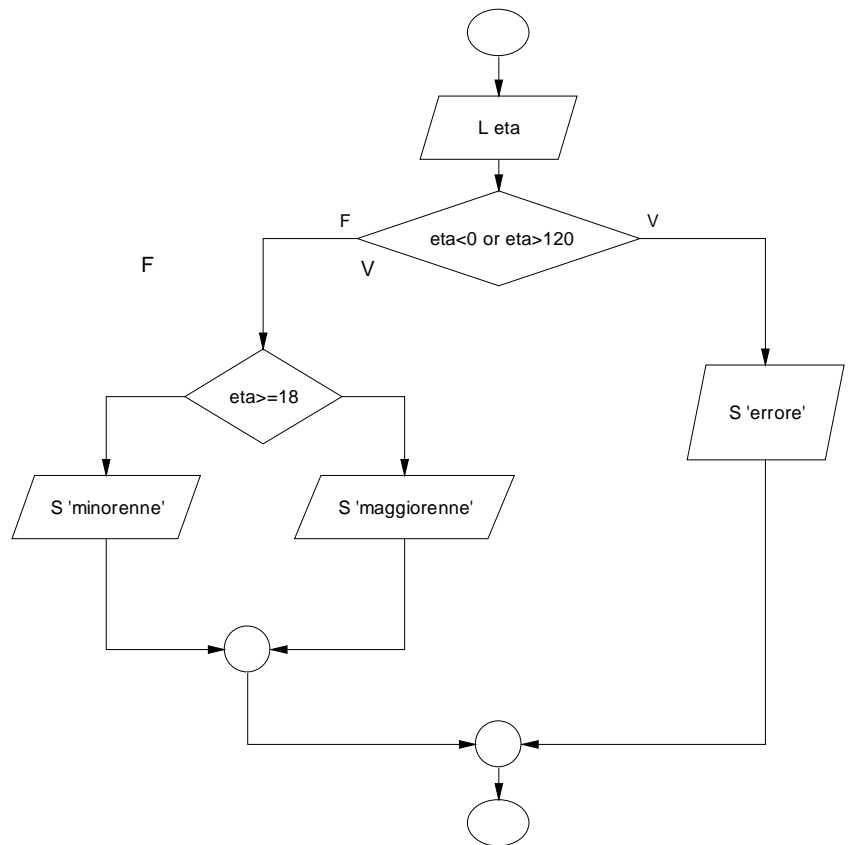
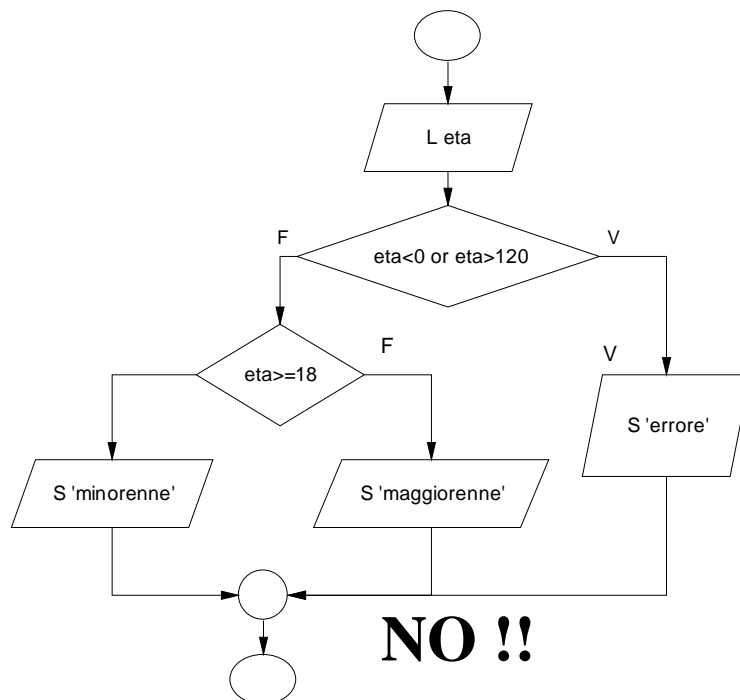
Esercizi risolti sulla struttura selettiva

SEL1*. Inserita un'età dire se siamo in presenza di un minorenne o di un maggiorenne*

NOTE: con un flow chart si vogliono di solito cogliere gli aspetti *essenziali* di un algoritmo; questo significa, ad esempio, 'trascurare' alcuni messaggi ovvi che nel programma in Pascal saranno invece indicati: è il caso del messaggio 'Inserire l'età' che dovrebbe precedere la sua lettura:

SEL2. *Inserita un'età', dire se siamo in presenza di un maggiorenne o di un minorenne; <u>controllare anche eventuali errori di inserimento da parte dell'utente</u>*

Prima soluzione: ipotizzando un età massima di 120 anni, controlliamo che l'età inserita non sia al di fuori dell'intervallo 0 (neonato) – 120.

NOTA: siate coerenti con l'utilizzo delle etichette 'V' e 'F': io ho scelto di proseguire per vero (V) sempre a destra e per falso (F) sempre a sinistra.

NOTA: è importante differenziare con un pallino-connettore separato le uscite dalle due strutture selettive. Il seguente flow chart è quindi <u>sbagliato</u>. Non riusciremmo infatti a tradurlo nel corrispondente programma Pascal (si accavallerebbero gli 'end' dei blocchi parti 'then' ed 'begin … end' delle 'else' dei due 'if' !!

NO !!

SEL2 bis. Con l'algoritmo precedente, quando l'utente sbaglia ad inserire un valore per l'età non si sa se per un valore negativo o maggiore di 120: si sa solo che ha commesso un errore ma non quale. Il seguente flow chart rappresenta un algoritmo che tiene conto anche di questo particolare, differenziando i controlli sui possibili errori:

Struttura a 'cascata'

L eta

F eta<0 V

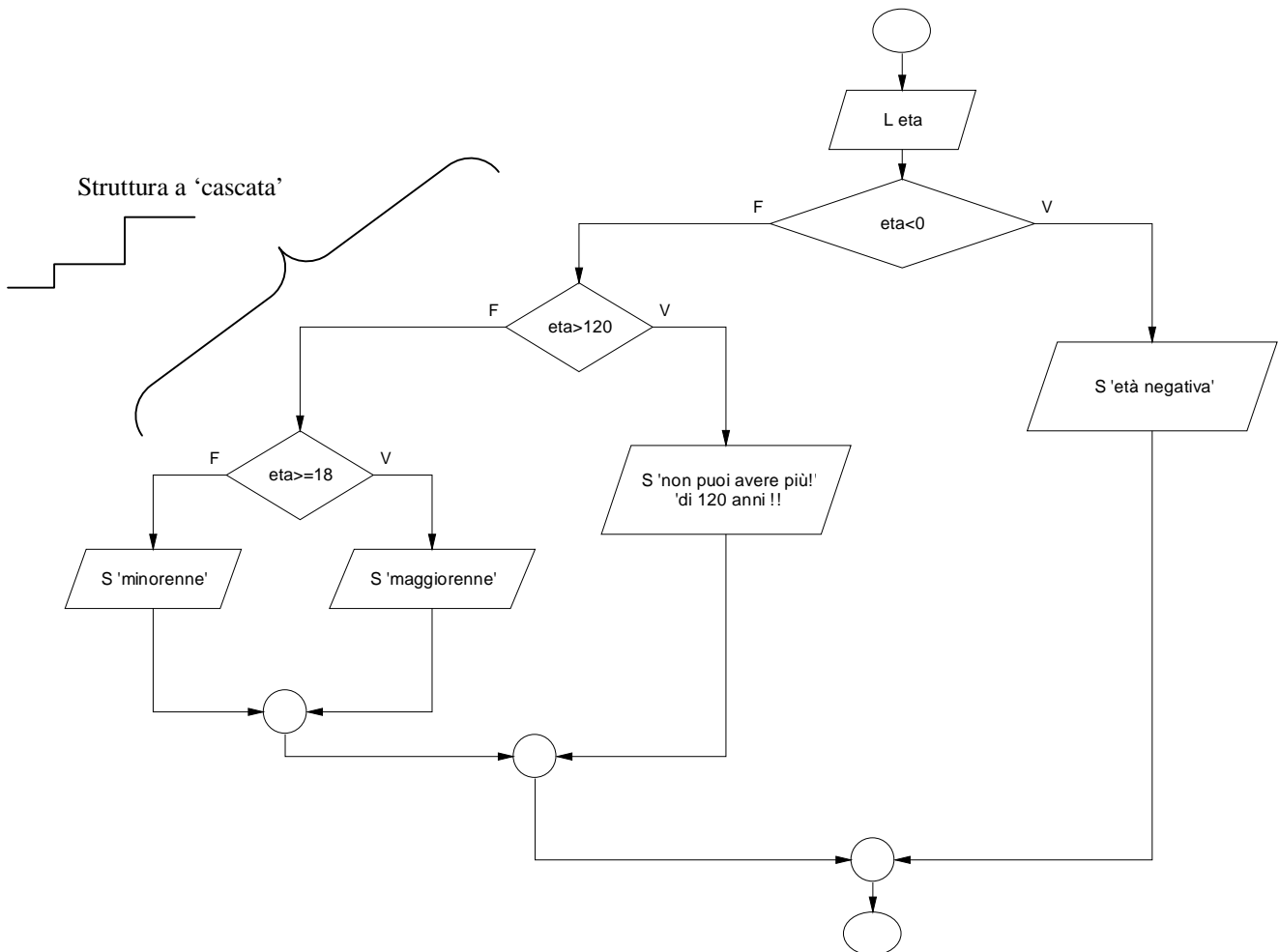
F eta>120 V

S 'età negativa'

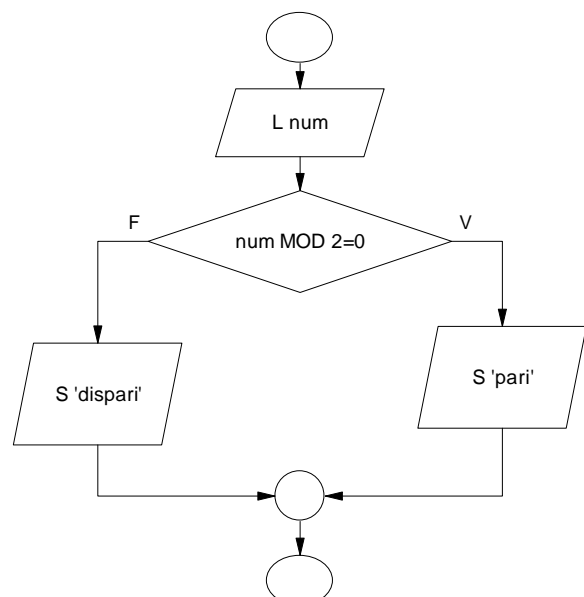
F eta>=18 V

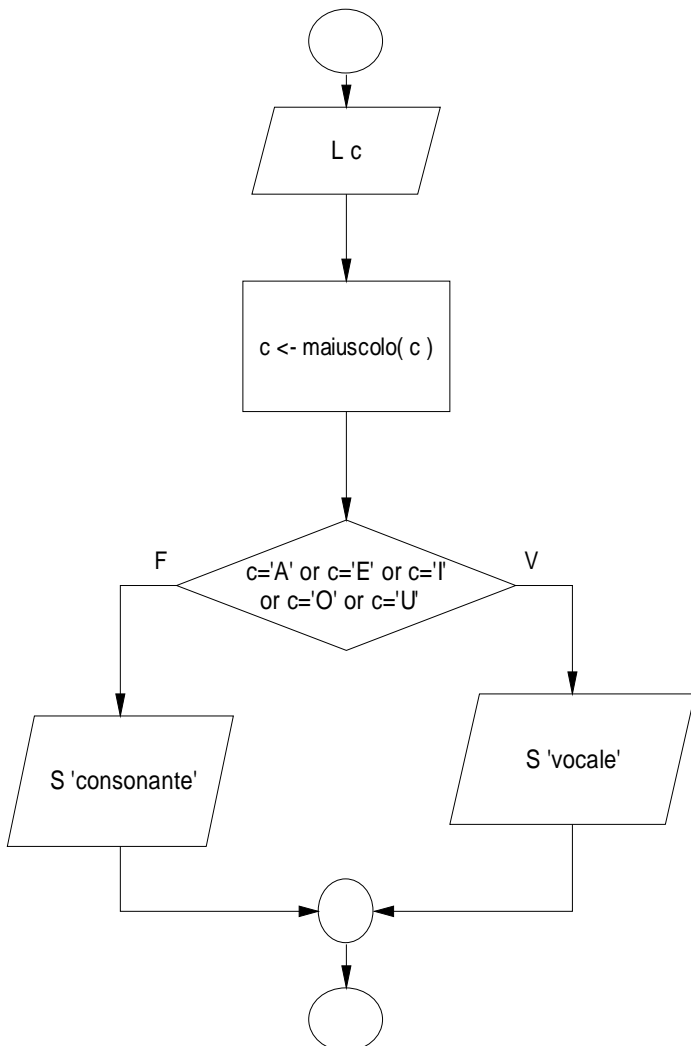
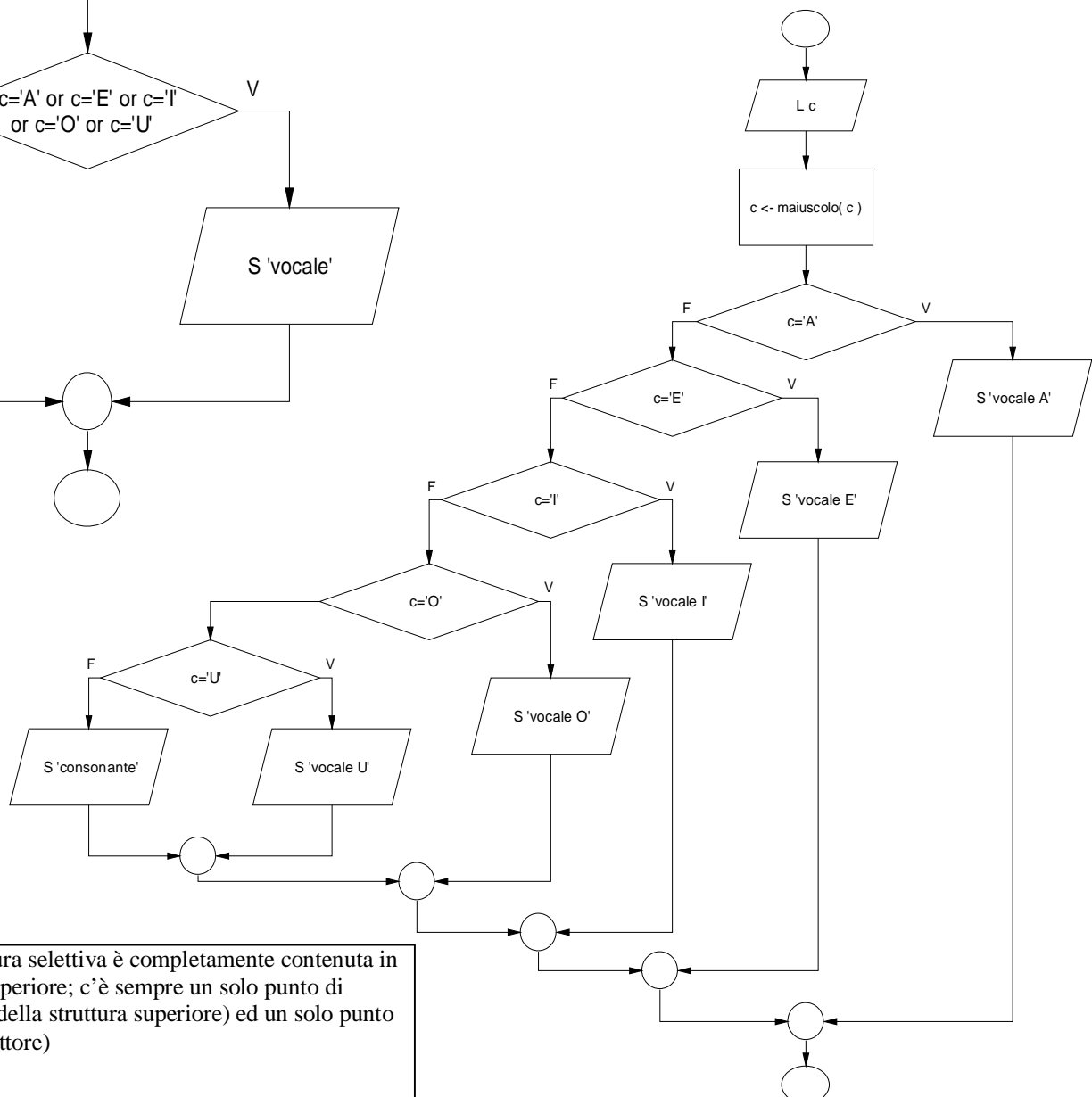
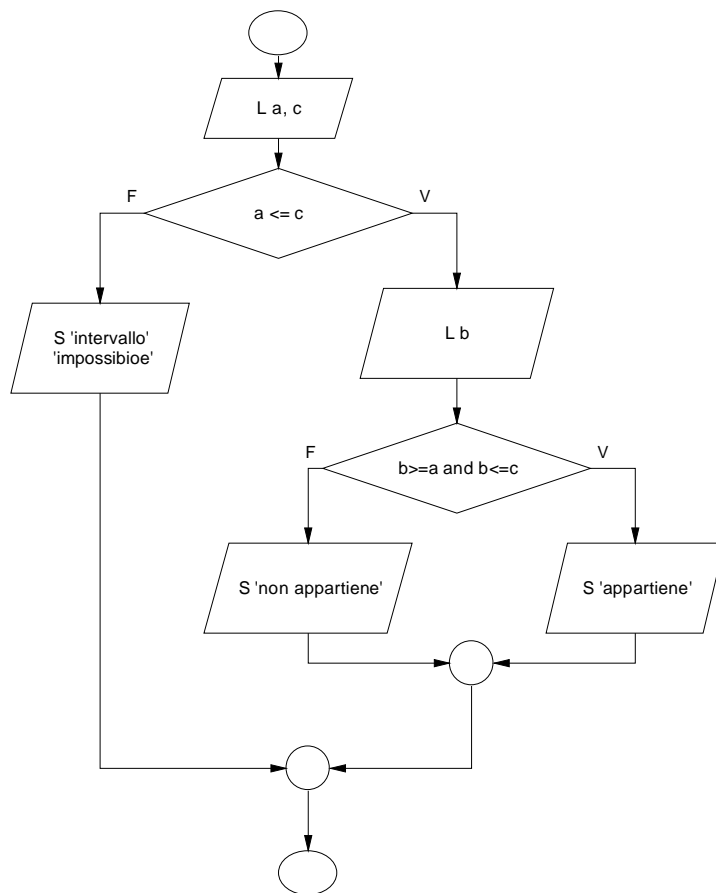
S 'non puoi avere più!' 'di 120 anni !!

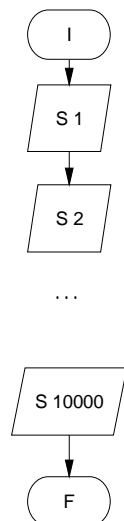
S 'minorenne'

S 'maggiorenne'

SEL3bis. Inserito un numero, dire se e' pari o dispari

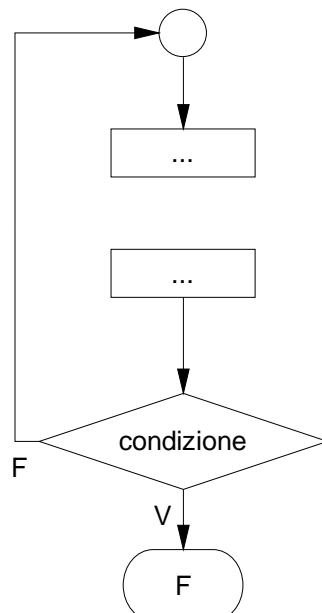
NOTA: si sta immaginando che in ogni linguaggio di programmazione esista un operatore per il calcolo del modulo: MOD.

E' comunque accettabile (mi sto rivolgendo ai miei alunni) usare gli operatori del Pascal.

SEL4. Inserito un carattere, dire se e' una vocale od una consonante.

Soluzione 1 (non si fa distinzione tra le vocali)

SEL5. Inseriti A, B e C dire se B e' compreso tra A e C; in pratica si controlla se B appartiene all'intervallo [A,C]

Iterazione (ripetizione, cicli)

Pur avendo a disposizione sequenza e iterazione rimangono molte le situazioni ‘intrattabili’ con questi strumenti. Consideriamo infatti questo problema all’apparenza molto semplice: stampare i numeri da 1 a 10000. Certamente la soluzione di usare per 10000 volte l’istruzione di scrittura per stampare ogni numero non è molto praticabile …

A parte i problemi di trovare uno spazio sufficiente per disegnare schemi così enormi ed il tempo per disegnarli, saremmo costretti a modificare sostanzialmente il diagramma al variare della richiesta (stampare solo i primi 5000 numeri o fino al 15000).

…

Partendo invece dalla considerazione che si stanno ripetendo istruzioni molto simili (cambia solo in valore da scrivere sul video) si perviene ad una struttura ciclica che fa ripetere una od un gruppo di istruzioni fino al raggiungimento di una condizione (in questo caso il raggiungimento del 10000).

Ecco lo schema generico (cioè non applicato alla soluzione di alcun problema particolare) di un flow chart per la struttura iterativa.

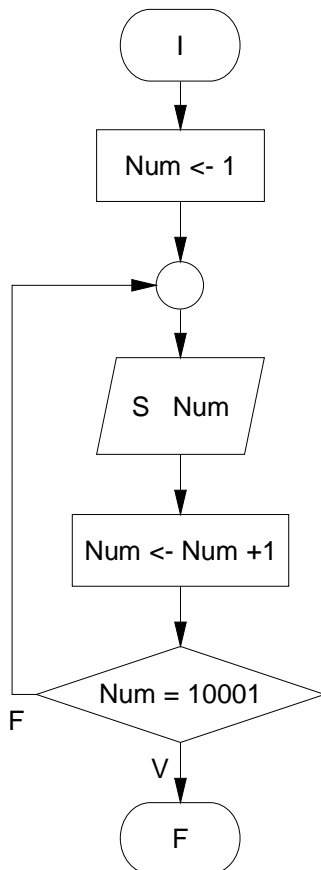
Le istruzioni sono ripetute una prima volta. Giunti alla condizione, se questa è vera allora si esce ed il ciclo termina. Se la condizione è falsa si ritorna all’inizio (connettore di ingresso).

Se all’interno del ciclo, prima o poi, una delle istruzioni farà sì che la condizione diventerà vera, il ciclo terminerà veramente.

Diversamente procederà all'infinito (come potrebbe servire per il controllo di un processo industriale).

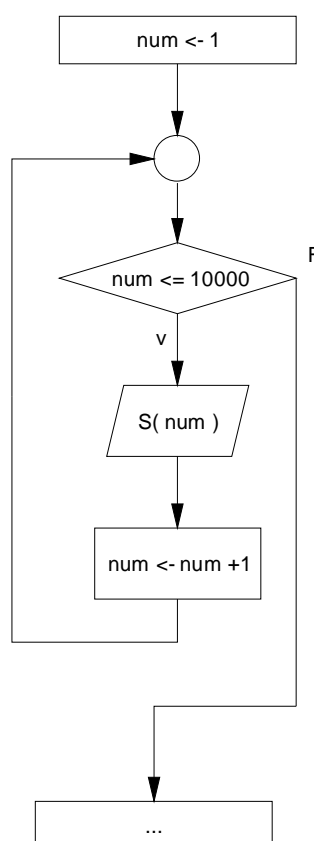
Questo tipo di ciclo è detto 'con ripetizione per falso', o a 'uscita per vero' ed ancora 'con controllo in coda' (cioè alla fine del ciclo).

Notiamo ancora: *con questo tipo di ciclo le istruzioni vengono eseguite almeno una volta*.

Vediamo il flow chart completo per la stampa dei numeri da 1 a 10000

La variabile *Num* è usata per controllare l'uscita dal ciclo. Essa viene impostata al valore 1 *prima* di iniziare il ciclo.

Il ciclo inizia stampando sul video il valore della variabile *Num* (la prima volta stamperà quindi 1).

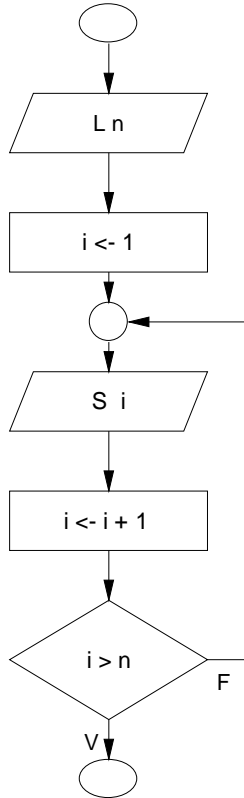
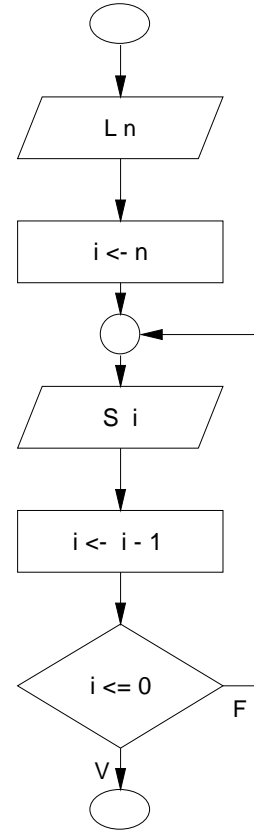
La variabile *Num* viene incrementata di 1.

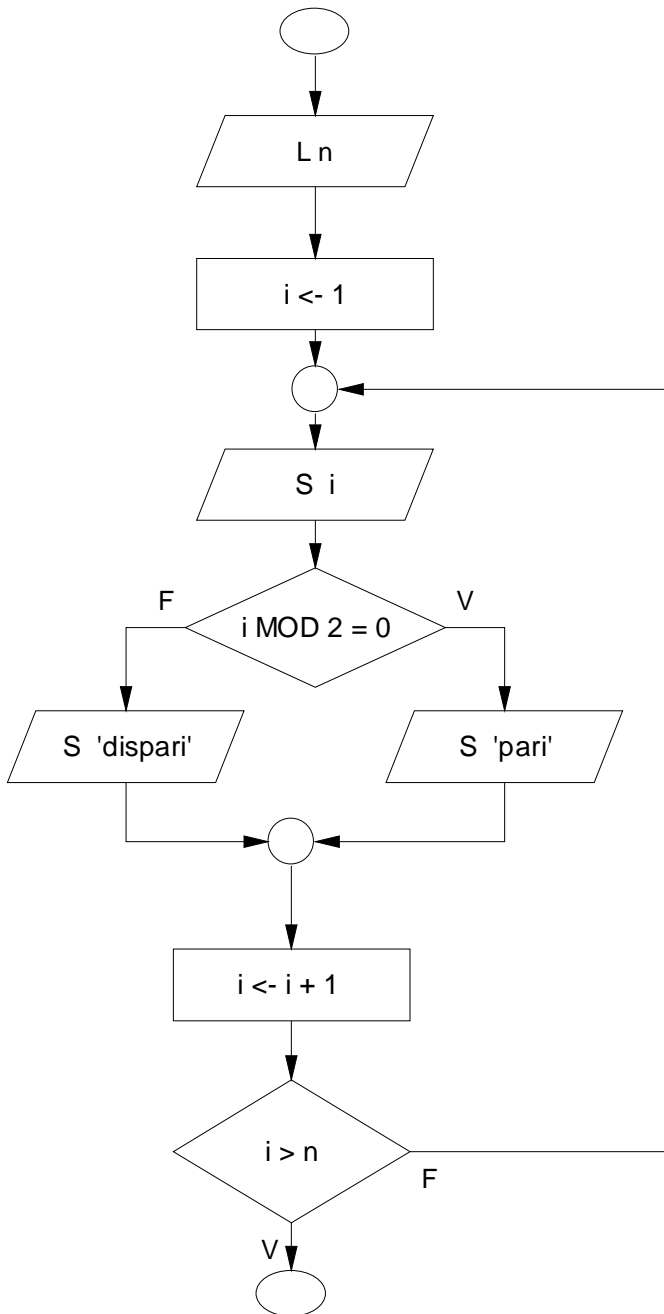
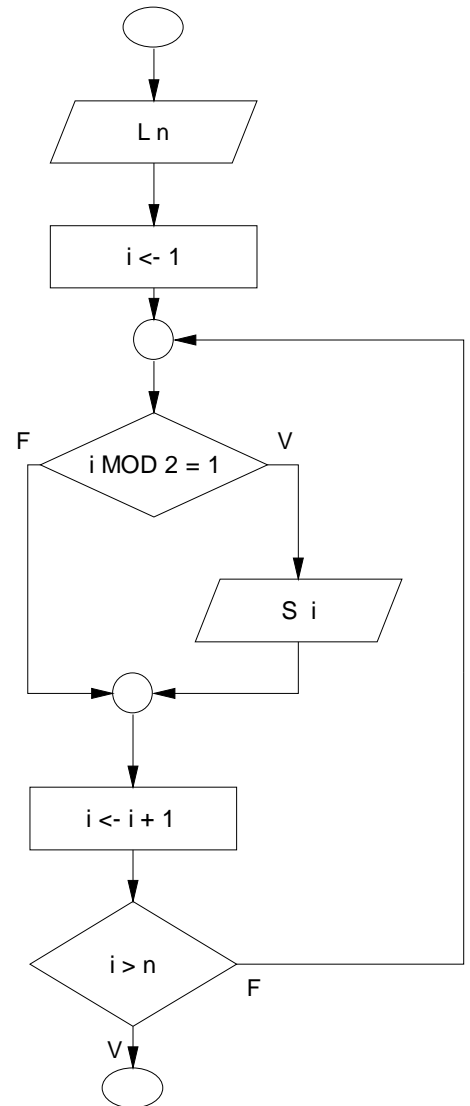
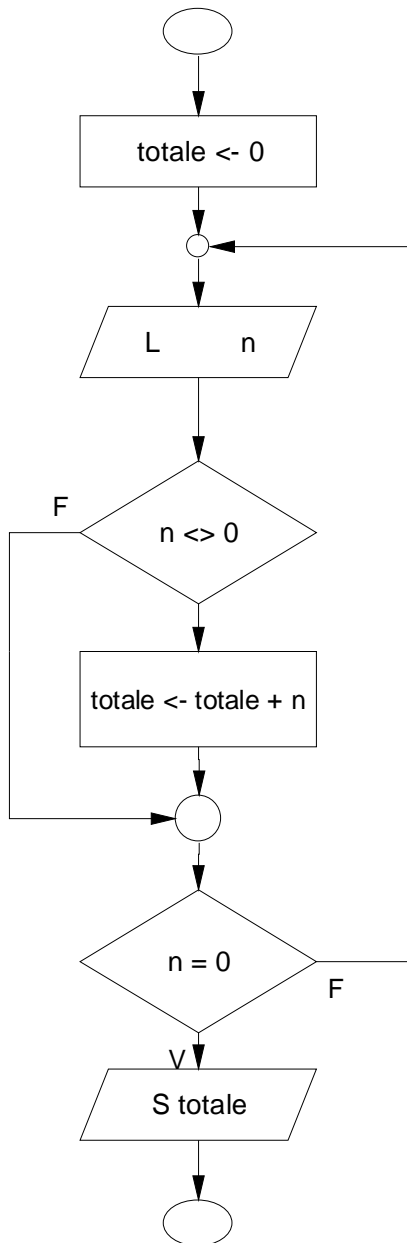
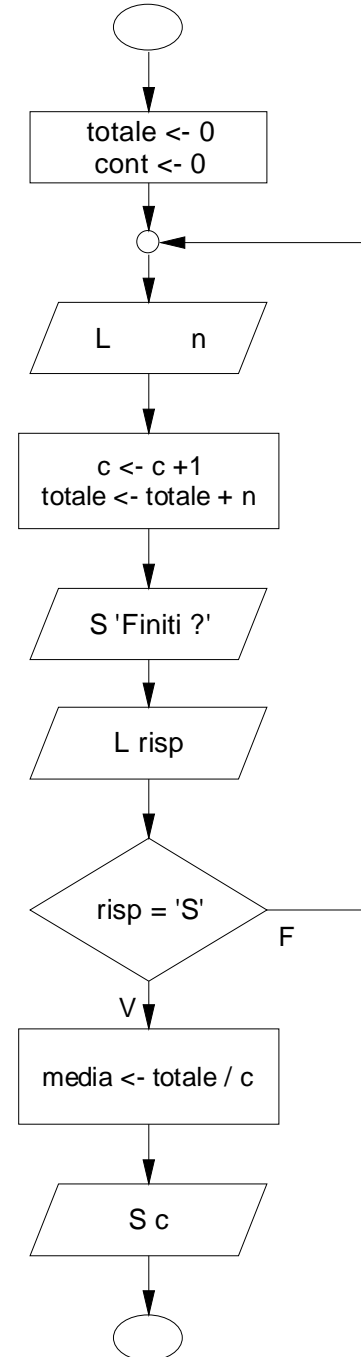
Se la variabile *Num* ha raggiunto il valore 10001 (quindi ha già stampato il 10000) si esce, altrimenti si ritorna all'inizio (verrà stampato sul video un nuovo numero, si incrementa ancora il valore di *Num* e così via …).

Ed ecco un esempio di flow chart per l'altro tipo fondamentale di ciclo, quello detto con controllo in testa (all'inizio) ed uscita per falso o, ancora, di ripetizione per vero.

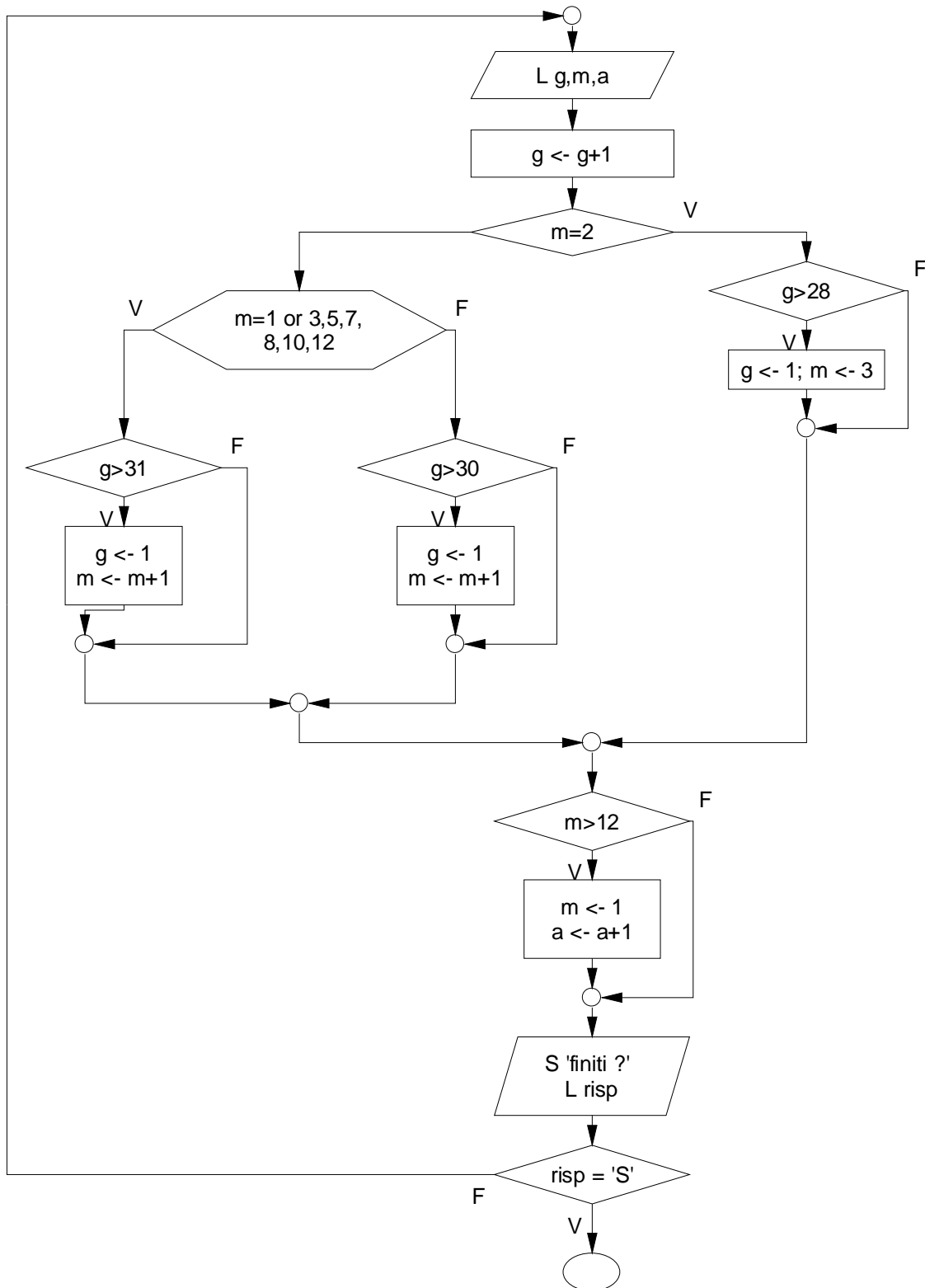
Infatti fintanto che la condizione rimane vera il ciclo viene ripetuto. Non appena la condizione diviene falsa il ciclo termina.

Il ciclo potrebbe anche non cominciare neppure, se la condizione fosse da subito falsa. Il ciclo visto in precedenza, invece, controllando la condizione alla fine, almeno una volta comunque esegue le istruzioni (e questo potrebbe essere indesiderabile quando si vogliono eseguire le istruzioni del ciclo solo se è vera fin da subito una certa condizione).

Esercizi risolti sulla struttura iterativa

ITE1: stampa dei primi N numeri naturali, con N letto da tastiera.

ITE2: stampa dei primi N numeri naturali, con N letto da tastiera; stampa dal più grande al più piccolo

ITE3. Stampa dei primi N numeri naturali, con N letto da tastiera; a fianco di ciascun numero indicare se e' pari o dispari

L n
i <- 1
S i
F
i MOD 2 = 0
V
S 'dispari'
S 'pari'
i <- i + 1
i > n
F
V

ITE4: Stampa dei numeri dispari minori o uguali a N, con N letto da tastiera

ITE5. Far inserire una sequenza di numeri da tastiera. La sequenza si intende terminata quando viene inserito il numero 0. Si deve calcolare e visualizzare la somma di tutti i numeri inseriti.

ITE6: Far inserire una sequenza di numeri da tastiera. E' l'utente del programma che dice quando sono finiti i numeri (rispondendo ad una opportuna domanda posta dal programma). Si deve calcolare e visualizzare la media di tutti i numeri inseriti.

NOTA: essendo il messaggio (domanda) per la terminazione del ciclo particolarmente importante, è meglio indicarlo nel flow chart.

1. Chiedere da tastiera l'inserimento di una sequenza di date. Ad ogni data, espressa nella forma giorno mese ed anno ('g', 'm' ed 'a' nel flow chart), aggiungere un giorno e stampare la data risultante. E' l'utente del programma che dice quando sono finite le date (rispondendo ad una opportuna domanda posta dal programma). Ipotesi semplificativa: considerare Febbraio sempre da 28 giorni. Traccia: è necessario prestare attenzione alle date che corrispondono a fine mese o anno: nella data risultante cambierà sicuramente il mese e forse anche l'anno …

2. Stampa dei numeri da B ad A con B>A.
Controllare che B sia maggiore di A
prima di iniziare.

L A

L B

B > A

F

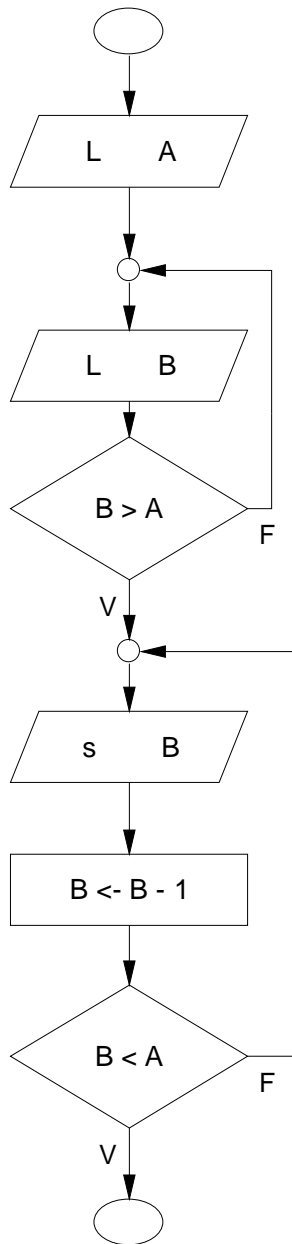
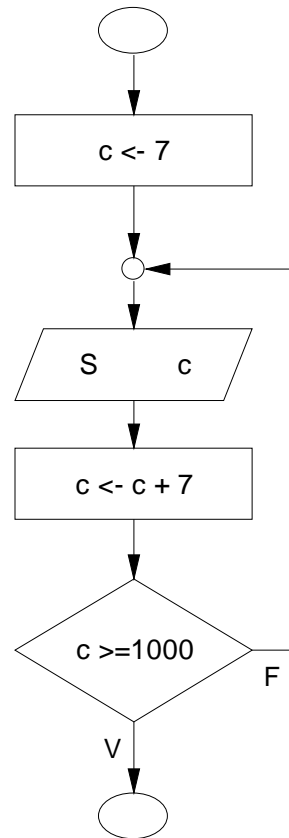
V

s B

B <- B - 1

B < A

F

V

3. Stampa dei multipli di 7 minori di 1000

I SOTTOPROGRAMMI - riutilizzare per sopravvivere !

Se un programmatore professionista dovesse sempre ricominciare da zero nello scrivere i programmi, diventerebbe presto un … accattone professionista ! K. Ad esempio, se tutte le volte che fosse necessario lo stesso calcolo complesso dovessimo riscrivere le sue istruzioni, i programmi diventerebbero più lunghi del necessario, più difficili da leggere, sprecheremmo più tempo e rischieremmo ogni volta di commettere errori diversi (e sempre per risolvere lo stesso problema).

Una pericolosa illusione …
L'utilizzo del 'copia/incolla' per ricopiare le righe farebbe risparmiare solo fatica ma ci esporrebbe a gravi rischi … Per convincercene, immaginiamo che le istruzioni siano una cinquantina e di averle copiate/incollate più volte in diversi programmi. Cosa accadrebbe se scoprissimo un errore nelle righe copiate così tante volte? Dovremmo modificare tutte le copie delle cinquanta istruzioni incriminate … e guai a dimenticarcene una !

La soluzione: i sottoprogrammi
Per fortuna una soluzione c'è e la sapete già usare. Lo avete fatto tutte le volte che avete usato comandi come sqrt(9) o clrscr. Questi comandi richiamano un programma secondario (si parla infatti di sottoprogrammi) che svolgono il compito previsto. Nel caso della sqrt il compito è il calcolo della radice quadrata di un numero (quello che deve essere specificato tra parentesi dopo il nome del comando); nel caso di clrscr non deve essere calcolato nulla ma semplicemente cancellato il video. Dopo l'esecuzione del comando il programma principale continua la sua elaborazione.

ESEMPIO (calcolo della radice quadrata di un numero)

```
Program prova;
var
  x: real;                   (* il numero di cui si vuole la radice quadrata *)
  radice_quadrata: real;     (* il risultato da calcolare *)

begin
  writeln('Inserisci un numero ed io calcolero'' la sua radice quadrata: ');
  readln(x);

  radice_quadrata:=sqrt(x);
```

> per soddisfare la richiesta della riga precedente il programma principale si 'ferma' e richiama il programma secondario sqrt; quando si riceve il risultato, il programma principale continua con la riga qui sotto stampando il valore calcolato *)

```
  writeln('risultato: ', radice_quadrata );
  readln;
end.
```

Tipi di sottoprogrammi: procedure e funzioni

Quando un sottoprogramma restituisce al programma principale un valore (come sqrt con la radice quadrata) si chiama funzione (function). Quando non restituisce nulla si chiama procedura (procedure). Un esempio di procedure è rappresentata dal comando clrscr: non restituisce nulla programma principale ma si limita a cancellare il video.

I parametri

Un sottoprogramma può aver bisogno di uno o più dati per portare a termine il suo compito. Ad esempio sqrt ha bisogno del numero di cui si vuole calcolare la radice. Clrscr non ha invece bisogno di alcun dato. Questi dati si chiamano parametri e vengono indicati tra parentesi dopo il nome del sottoprogramma.

Nel comando sqrt(9) il parametro è costituito dalla costante numerica 9. Nel comando sqrt (12+3) il parametro è costituito dal risultato dell'espressione 12+3. Nel comando sqrt(X) il parametro è il valore contenuto nella variabile X al momento dell'uso del comando.

Se i parametri sono più d'uno vanno separati con la virgola. Ad esempio il comando gotoxy(10, 5) sposta il punto di scrittura sul video alla colonna 10, riga 5. Il primo parametro è il numero della colonna, il secondo il numero della riga. NOTA: per provare il comando gotoxy dovete indicare uses crt come spiegato alla pagina successiva.

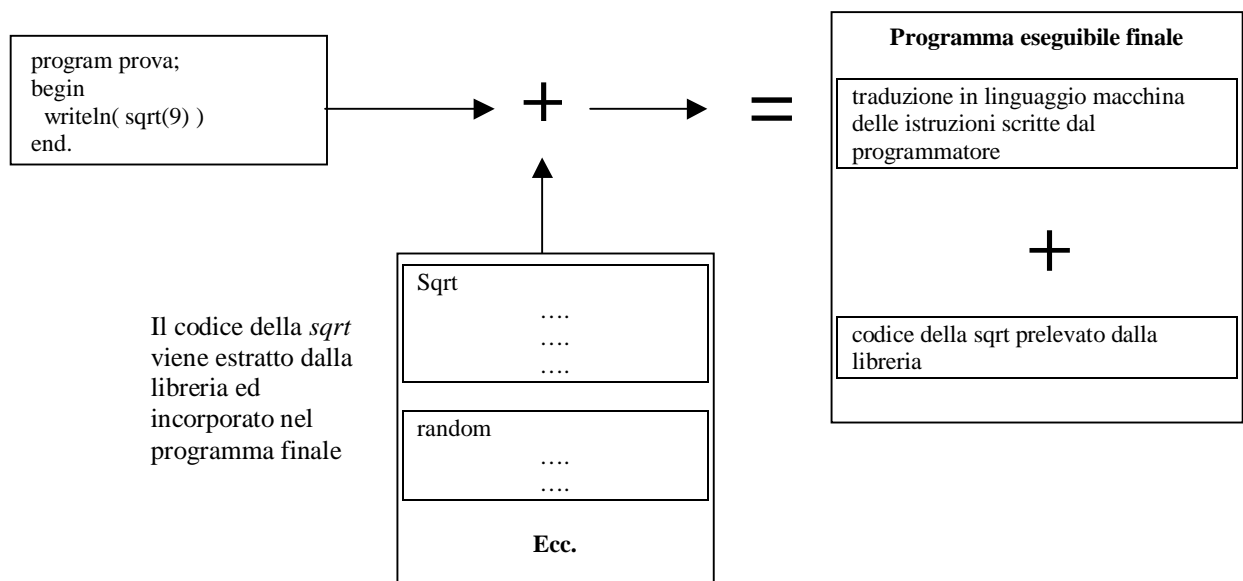
Interessante... ma quali e quanti sottoprogrammi abbiamo a disposizione ?

Abbiamo due possibilità:

Usare un sottoprogramma scritto da altri. I sottoprogrammi riutilizzabili sono spesso raccolti in librerie. Una libreria è un file (archivio) normalmente presente sul disco e contiene blocchi di codice già tradotto in linguaggio macchina che possono essere a comando incorporati in un programma. Alcune librerie sono sempre fornite insieme all'ambiente di sviluppo che si decide di usare (sia che si tratti di un prodotto commerciale sia che si tratti di un prodotto gratuito come accade per l'open source).

Ad esempio con il Turbo Pascal sono disponibili un buon numero di librerie insieme al prodotto acquistato. Il Turbo Pascal chiama le librerie unit (unità di programmazione). NOTA: per un elenco completo di queste librerie consultate l'help in linea del turbo Pascal (lo potete attivare con la combinazione SHIFT-F1, cercando poi sotto la lettera S il paragrafo Standard Unit).

La funzione sqrt ed altri sottoprogrammi di uso assai comune sono memorizzate in una libreria (la System) per la quale non è neppure necessario indicarne l'uso: viene automaticamente inclusa con il resto del programma.

Il comando clrscr e gotoxy() invece sono due dei sottoprogrammi disponibili nella unit crt. Per questa ed altre unit (ed in particolare per tutte quelle create ed aggiunte dai programmatori) è invece necessario indicare il loro nome nella sezione uses del programma. Esistono unit con sottoprogrammi per fare grafica (graph) e TANTE altre cose che scopriremo insieme un poco alla volta J.

In commercio sono poi disponibili parecchie librerie per i compiti più svariati. Se una libreria è fatta bene vale mille volte i soldi che costa: sviluppare software costa MOLTO! Su Internet è facile trovare ed acquistare queste librerie.

Su internet, soprattutto sui siti dedicati alla programmazione si trovano anche MOLTE librerie gratuite.

Scriversi i sottoprogrammi da soli raggruppandoli poi in librerie (che possono essere scambiate, rese disponibili gratuitamente su Internet o vendute). Un sottoprogramma viene scritto di solito prima all'interno di un programma normale e solo in un secondo tempo viene aggiunto ad una libreria.

Come si fa in Turbo Pascal ad usare un sottoprogramma di una libreria?
Prima della sezione VAR del programma si mette una sezione intitolata USES e si elencano, separati da virgole, i nomi delle unit che contengono i sottoprogrammi che si vogliono utilizzare. Come dicevo prima, i sottoprogrammi clrscr e gotoxy() sono contenuti nella unit crt e quindi per usarli:

```
program prova;

uses  crt;

begin
  clrscr;
  gotoxy(10,5);
  writeln('sto scrivendo alla colonna 10, quinta riga dello schermo ...);
  readln;
end.
```

Prima di esaminare in dettaglio come si scrivono i sottoprogrammi con il turbo Pascal dovete essere convinti della loro utilità.

Riassunto dei benefici dell'uso dei sottoprogrammi:

Risparmio di tempo e di spazio: le istruzioni che corrispondono al sottoprogramma sono scritte una volta sola, tradotte ed inserite in una libreria; il programmatore non le deve riscrivere (o copiare/incollare) ma può richiamarle semplicemente scrivendo il nome del sottoprogramma ed indicando tra parentesi le costanti o le variabili che contengono i valori necessari al funzionamento del sottoprogramma (i parametri).

Non incorre nei rischi del copia/incolla:

se viene scoperto un errore nel sottoprogramma: si modificano solo le istruzioni di quest'ultimo e si aggiorna poi la libreria con la nuova versione; è poi sufficiente ricompilare/linkare i programmi che ne facevano uso (questi ultimi non sono stati modificati!);

se scritto in modo corretto il sottoprogramma non interferisce con il resto del programma: un sottoprogramma può avere le sue variabili per i cicli e per tutto il resto; ha, in definitiva, un suo spazio di lavoro che lo rende indipendente; riceverete maggiori dettagli più avanti;

Rende il programma più comprensibile: è palese come il codice sulla destra, a differenza di quello a sinistra, comunichi al lettore immediatamente lo scopo per cui è stato scritto:

```
const
   pi_greco=3.14;
var
  x: real;
begin
  writeln( 4/3 * pi_greco * 12 * 12 * 12)
end.
```

```
writeln( VolumeSfera(12) )
```

Infatti pochi sanno riconoscere nella formula a sinistra il calcolo del volume di una sfera di raggio 12 ... Nel riquadro a destra è invece palese che si sta invocando una funzione dal nome molto chiaro ed è altrettanto evidente che il valore comunicato tra parentesi è il raggio della sfera. In ogni caso le modalità d'uso di ogni sottoprogramma sono documentate nel manuale tecnico che accompagna sempre una libreria.

Diminuisce la complessità della scrittura dei programmi: un programma può essere assemblato a partire da sottoprogrammi più semplici da trattare. E' anche più facile trovare gli errori concentrandosi solo sulle righe di un sottoprogramma per volta. Inoltre rende possibile sviluppare uno stesso programma insieme ad altri programmatori senza che questi interferiscano tra loro: ad ogni programmatore viene assegnato lo sviluppo di un certo numero di sottoprogrammi che vengono poi fusi assieme.

Nessun team di sviluppo si sognerebbe oggi di sviluppare un progetto software senza adottare questo tipo di programmazione detto modulare. Questi concetti sono stati poi ampliati nella programmazione ad oggetti (OOP) che sarà materia di studio in quarta/quinta.

Mentre vengono sviluppati e provati i sottoprogrammi vengono di solito scritti direttamente all'interno di un programma che li usa. È solo in un secondo momento, quando il sottoprogramma è stato perfettamente testato, di quest'ultimo viene inserito in una libreria.

Qui a lato ho evidenziato come in un programma i sottoprogrammi vengono scritti dopo la sezione VAR e devono terminare prima del begin di inizio programma. Non c'è limite al numero di sottoprogrammi che si possono scrivere (se non quelli imposti dall'ambiente di sviluppo per la scrittura del programma principale).

Come dicevo, questo facilita la scrittura dei sottoprogrammi ma ne riduce la facilità di riutilizzo (dovremmo ricorrere ad un copia/incolla per usarli in altri programmi), pur mantenendo inalterati tutti gli altri benefici. Una volta che il sottoprogramma è stato ben testato lo si può togliere dal programma principale ed includerlo in una libreria, recuperando *tutti* i benefici discussi prima.

Anche noi procederemo così, visto che l'obiettivo è imparare a scrivere i sottoprogrammi.

NOTA: nel programma principale possiamo sfruttare un qualsiasi sottoprogramma in più punti, tutte le volte che ciò si rende necessario!

```
program prova;

const ...
var ...

sottoprogramma 1

sottoprogramma 2

Begin

    ... uso sottoprogramma1

    ... uso sottoprogramma1

    ... uso sottoprogramma2

    ....

End.
```

Come si scrive un sottoprogramma ?

Iniziano con la parola procedure o function seguita dal nome che il programmatore vuole dare al sottoprogramma. Se non sono previsti parametri si mette il punto e virgola subito dopo il nome del sottoprogramma (tutti i nostri esempi iniziali non avranno parametri per semplicità ...). Poi tra begin e end si mettono le istruzioni che si vogliono far eseguire quando si richiama il sottoprogramma. Ad esempio, ecco come si scrive una procedure per stampare sullo schermo un rettangolo fatto da asterischi.

```
program esempio;
uses crt;
```

```
procedure
disegnaQuadrato;
begin
  writeln('****');
  writeln('****');
  writeln('****');
  writeln('****');
end;

begin
  disegnaQuadrato;
  writeln;
  writeln;
  disegnaQuadrato;
  writeln;
  writeln;
  readln

end.
```

Il sottoprogramma viene dichiarato una volta sola e richiamato due volte nel programma principale.

Fosse necessario stampare anche mille rettangoli, le istruzioni del sottoprogramma sono state scritte una volta sola.

Se si vogliono aumentare o diminuire il numero di righe o colonne degli asterischi è sufficiente farlo solo nel sottoprogramma ed il nuovo funzionamento si ripercuoterà automaticamente in TUTTO il programma.

Nel prossimo esempio scriveremo un sottoprogramma che restituisce un numero scelto a caso per il gioco del lotto (un numero da 1 a 90). Poiché restituisce un valore dobbiamo usare una function, che si differenzia leggermente da una procedure, ma non di molto.

```
program esempio;

function estraiLotto : integer;
begin
  randomize;
  estraiLotto := random(90) + 1
end;

begin
  writeln ('primo numero lotto estratto: ');
  writeln ( estraiLotto );

  writeln ('secondo numero lotto estratto: ');
  writeln ( estraiLotto );
 end.
```

Ho evidenziato le 2 cose che cambiano rispetto ad una *procedure*.

1. Dopo il nome del sottoprogramma (e dopo la parentesi chiusa dell'eventuale lista dei parametri che nell'esempio manca) si mettono i due punti ed il tipo del valore che viene restituito.

2. La *function* DEVE terminare con l'assegnazione al suo nome del valore da restituire a chi usa la *function*.

Sottoprogrammi con uso di parametri

La funzione estraiLotto non può essere utilizzata per altre simulazioni; ad esempio per un gioco di dadi (dove, ovviamente, l'intervallo di valori e quello che va da uno a sei). Possiamo però riscrivere la funzione in modo da poter indicare al momento della chiamata il massimo numero da estrarre. Cambiamo anche il nome della funzione in quanto sarebbe fuorviante lasciare il riferimento al lotto.

```
program esempio;

function estrai(Massimo: integer) : integer;
begin
  randomize;
  estraiLotto := random(massimo) + 1
end;

begin
  writeln (‘primo numero lotto estratto: ‘);
  writeln ( estrai(90) );

  writeln (‘tiro il dado ... è uscito: ‘);
  writeln ( estrai(6)  );

end.
```

I parametri vengano specificati tra parentesi prima del nome del sottoprogramma. Per ciascuno deve essere specificato il tipo. Notate come sia ancora necessario, se si tratta di una funzione, indicare il tipo del valore restituito dal sottoprogramma.

Il nome che scegliamo per il parametro serve al sottoprogramma per sapere come riferirsi al valore che riceve quando viene chiamato.

```
program esempio;
var numero: integer;

procedure asterischi(quanteRighe: integer);
var i: integer;
begin
 for i:=1 to quanteRighe do
   writeln(‘***************’);
end;

begin
  writeln (‘quante righe di * devo stampare? ‘);
  readln(numero);

  asterischi( numero );
  readln;
end.
```

Anche le procedure possono ricevere valori sotto forma di parametri. Nell'esempio qui a lato la procedura riceve il numero di righe di asterischi che deve stampare (il parametro chiamato *quanteRighe*).

Essendo una procedura, dopo la parentesi determina l'elenco dei parametri non deve essere specificato un tipo come abbiamo visto per le funzioni.

In questo esempio c'è un altro particolare molto interessante da notare: dopo l'intestazione con il nome della procedura/funzione è possibile iniziare una sezione VAR in cui dichiarare le variabili ad uso esclusivo del sottoprogramma. Nel caso in questione si tratta della variabile di controllo del ciclo for.

Torneremo presto su questo argomento.

Parametri per valore (by value)

```
program esempio;
var numero: integer; simboli: string;

procedure asterischi(quanteRighe: integer; riga: string);
var i: integer;
begin
 for i:=1 to quanteRighe do
   writeln(riga);
end;

begin
  writeln ('quante righe di * devo stampare? ');
  readln( numero );

  writeln ('che simboli uso per la stampa? ');
  readln( simboli);


  asterischi( numero, simboli );
  readln;
end.
```

Qui a lato trovate una versione potenziata della procedura esaminata in precedenza. Chi la usa non è più limitato ad una riga di asterischi ma può scegliere, specificandolo come secondo parametro, la riga di caratteri da usare.

Quando ce n'è più d'uno, i parametri devono essere separati con un punto e virgola e per ciascuno deve essere specificato il tipo corrispondente.

NOTATE: mentre quando scriviamo il testo del sottoprogramma gli eventuali parametri devono essere separati con un punto e virgola, quando invece il sottoprogramma viene usato i valori o le variabili che corrispondono ai parametri previsti vengono invece separati con una virgola.

```
program esempio;
var num1, num2: integer;

function max(a,b: integer) : integer;
begin
  if a>=b then
    max:=a
  else
    max:=b
end;

begin
  writeln ('inserisci un numero');
  readln ( num 1 );

  writeln ('inserisci un altro numero');
  readln ( num 2 );

  writeln ('il maggiore dei numeri inseriti è: ');
  writeln ( max(num1, num2)  );

   readln;
end.
```

La funzione qui a lato restituisce il più grande dei due valori che riceve come parametri. Essendo questi ultimi dello stesso tipo, è possibile dichiararne il tipo insieme (separando però i loro nomi con una virgola)

IMPORTANTE: i nomi delle variabili che eventualmente vengono utilizzate al momento del richiamo del sottoprogramma non devono chiamarsi per forza come i parametri. Detto in altre parole: per il fatto di aver chiamato *a* e *b* i parametri della funzione non siamo assolutamente obbligati a chiamare allo stesso modo le variabili del programma principale usate per chiamare il sottoprogramma; nell'esempio, le variabili dichiarate nel programma principale si chiamano infatti *num1* e *num2*: quello che accade è che il valore contenuto in *num1* viene copiato nel parametro *a* e che il valore contenuto in *num2* viene copiato nel parametro *b*. Si parla infatti di **passaggio dei parametri per valore** (**by value**).

NOTA. Se vi state domandando come si faccia a distinguere la modalità di passaggio dei parametri *by value* dall'altra che esamineremo tra poco, considerate questa semplice 'regola': quando nella specifica del parametro appare solo il suo nome ed il suo tipo il passaggio è *by value*. Dovremo infatti aggiungere qualche cosa nella dichiarazione per scegliere l'altra modalità.

Caratteristiche del passaggio dei parametri per valore

Il passaggio per valore ha una caratteristica decisamente interessante: è ‘sicuro’ in quanto non consente al sottoprogramma di modificare, tramite un parametro, una variabile del programma principale. Cerchiamo di capire il perché con un esempio.

```
program esempio;
var numero: integer; simboli: string;

procedure asterischi(quanteRighe: integer; riga: string);
begin
  repeat
    writeln(riga);
    quanteRighe := quanteRighe – 1;
  until quanteRighe=0;
end;

begin
  numero:=5;
  simboli:='******************';

  asterischi( numero, simboli );

  writeln( numero ); (* che valore viene stampato ? *)
  readln;
end.
```

Questa volta nel sottoprogramma il ciclo è realizzato con la struttura *repeat ... until*. Ma attenzione: quest’ultimo usa come contatore il parametro *quanteRighe* stesso, diminuendolo di uno ad ogni ciclo.

La domanda da un milione di euro è: se la variabile *numero* prima di chiamare la procedura vale 5, dopo aver chiamato la procedura (che apporta modifiche al suo primo parametro) quale sarà il suo valore?

Risposta: lo stesso che aveva prima della chiamata; detto in altre parole, **il passaggio dei parametri *by value* non consente ad un sottoprogramma di modificare il valore di una variabile esterna utilizzata nel programma principale per richiamare la procedura stessa**.

I parametri indicati nella procedura sono chiamati parametri formali; quelle indicate nel programma al momento dell'utilizzo del sottoprogramma sono invece chiamati parametri attuali.

```
program esempio;
var numero: integer; simboli: string;

                        Parametri formali
procedure asterischi(quanteRighe: integer; riga: string);
var i: integer;
begin
 for i:=1 to quanteRighe do
   writeln(riga);
end;

begin
  writeln ('quante righe di * devo stampare? ');
  readln( numero );

  writeln ('che simboli uso per la stampa? ');
  readln( simboli);

             Parametri attuali
  asterischi( numero, simboli );
  readln;
end.
```

Il passaggio è by value: il sottoprogramma non può modificare il valore dei parametri attuali numero e simboli.

Il valore che la variabile del programma numero ha al momento dell'uso del sottoprogramma viene assegnato alla variabile parametro formale corrispondente (quanteRighe). La stessa cosa avviene per il secondo parametro. Di fatto il sottoprogramma opera su una copia dei parametri attuali e della copia può disporre come preferisce senza alterare realmente il valore delle variabili

Se la procedura potesse invece modificare (senza che il suo utilizzatore ne fosse consapevole) le variabili che corrispondono ai parametri attuali si correrebbe il rischio di incappare in errori molto difficili da scoprire.

Immaginiamo infatti la situazione in cui la procedura asterischi, una volta sperimentata e considerata 'perfetta', venga tolta come codice dal programma visto prima ed inserita in una libreria chiamata stampe. Il programma apparirebbe allora così strutturato:

```
program esempio;
uses stampa;
var numero: integer; simboli: string;

begin
  numero:=5;
  simboli:='******************';

  asterischi( numero, simboli );

end.
```

Indico l'uso della libreria

Come vedete, le istruzioni della procedura asterischi non sono più visibili.

Il codice corrispondente, già compilato in linguaggio macchina, viene incorporato nell'eseguibile finale dal linker.

Nel programma rimangono solo le righe

Il programmatore non ha modo di capire se la procedura asterischi modificherà o meno il valore di uno dei suoi parametri formali! Ed anche se il codice dal sottoprogramma fosse ancora inserito direttamente nel programma saremo sempre costretti ad un alto livello di attenzione perdendo molto tempo nel controllare le istruzioni di ogni sottoprogramma per capire se esiste questa possibilità.

Insomma, sarebbe un bel guaio se il sottoprogramma potesse modificare (senza che questo fosse dichiarato in qualche modo) a piacimento il valore di uno dei parametri attuali (numero, simboli)!

Invece, grazie ai vincoli imposti dal passaggio dei parametri per valore (by value) il programmatore è sicuro che un sottoprogramma neanche per errore potrà modificare il valore di un parametro attuale.

Esistono comunque dei vincoli tra i parametri indicati ne la scrittura del codice di un sottoprogramma (chiamati parametri formali) e quelli che il programmatore utilizza nel programma principale quando intende usare il sottoprogramma (chiamati parametri attuali):

per ogni parametro formale deve essere indicato al momento dell'utilizzo del sottoprogramma un corrispondente parametro attuale; riferendoci all'esempio della procedura asterischi, non è ammesso il suo utilizzo specificando solo il numero delle righe o solo i singoli da utilizzare; non ha ovviamente senso anche cercare di usare più parametri attuali di quelli previsti;

il parametro attuale dev'essere dello stesso tipo del parametro formale o almeno compatibile; il parametro formale quanteRighe della procedura asterischi è di tipo integer, per cui, richiamandola, come primo parametro attuale potremo solo indicare o una costante integer o una variabile integer o una qualsivoglia espressione che restituisca un valore integer;

se un parametro formale fosse di tipo real allora il corrispondente parametro attuale potrebbe essere al limite anche un integer visto che quest'ultimo tipo di dato è compatibile con il real (ad esempio il 5, integer, verrebbe convertito nel real 5.0); il contrario non sarebbe invece possibile, in quanto il numero di byte necessario a rappresentare un real è maggiore di quello necessario a rappresentare un integer; ad esempio, il numero real 12.67 non viene automaticamente troncato al valore integer 12 a causa di una perdita di precisione che potrebbe risultare inaccettabile; il programmatore può però indicare come parametro attuale l'espressione trunc(12.67) che dietro esplicito comando 'tronca' un valore real in un integer (in realtà in un longint, compatibile a sua volta con un integer a patto che il valore cada nell'intervallo previsto per gli integer);

i parametri attuali devono essere forniti nello stesso ordine logico previsto per quelli formali; la procedura asterischi si aspetta il numero delle righe come primo parametro e la stringa dei singoli da usare come secondo; tentare di invertire l'ordine non ha gravi conseguenze, in questo particolare caso, perché i tipi dei due parametri sono diversi ed il compilatore si accorgerebbe subito del problema e bloccherebbe la compilazione con un messaggio d'errore; ben diverso è il caso di parametri dello stesso tipo: il compilatore non s'accorgerebbe di nulla dal momento che dal suo punto di vista si tratterebbe comunque di due interi o di due stringhe eccetera

Viva l'indipendenza!
Quella che sto per enunciare non è una regola sintattica ma una dettata dall'esperienza. Idealmente un sottoprogramma dovrebbe essere strutturato in modo da essere il più indipendente possibile dal programma che lo utilizzerà o da altri sottoprogrammi.

Questo obiettivo viene raggiunto soprattutto con l'utilizzo dei parametri, evitando di usare direttamente eventuali variabili del programma principale. Consideriamo infatti questo esempio (ho ripreso la forma più semplice della procedura asterischi):

```
program esempio;
var i: integer;

procedure asterischi(quanteRighe: integer);
begin
 for i:=1 to quanteRighe do
   writeln('***************');
end;

begin
  writeln ('quante righe di * devo stampare? ');
  readln(numero);

  asterischi( numero );
  readln;
end.
```

La modifica che ho apportato è piccola ma sostanziale: ho spostato la dichiarazione della variabile *i* utilizzata per il ciclo *for* dall'interno della procedura alla sezione *var* del programma principale.

Se proviamo a mandare in esecuzione il programma tutto sembra funzionare e, in effetti, funziona veramente... Ma la procedura è in realtà meno leggibile, meno riutilizzabile e meno sicura.

Meno leggibile: non è sufficiente passare in rassegna tutte le istruzioni della procedura per capire tutto della procedura; infatti per capire cosa sia la i usata dal ciclo for dobbiamo cercare nel programma principale la sua dichiarazione; e se qualcuno tra voi sta pensando che realtà è ovvio che la i sia una variabile di tipo integer, vi invito a provare il seguente programma!

```
program prova;
var i: char;
begin
  for i:='A' to 'Z' do
     write( i );
  readln;
end.
```

Non è più così ovvio che la variabile usata per un ciclo *for* sia per forza di tipo integer, vero ???

Meno riutilizzabile: se spostiamo la procedura in un altro programma o la includiamo in una libreria non è detto che in quel programma o in quella libreria sia presente, come nelle programma originale, una variabile di tipo integer chiamata i; il ciclo for della procedura non potrebbe quindi funzionare. E per quanto vi possa sembrare strano questo è il caso più fortunato tra quelli che possono capitare: infatti il compilatore ci avverte dell'assenza della variabile ed il programmatore può intervenire e soprattutto si accorge che c'è un problema.

Meno sicura: provate a pensare se nel programma in cui viene copiata la procedura esiste già una variabile con lo stesso nome e dello stesso tipo: il compilatore non farebbe una grinza perché la procedura pretende una certa variabile nel programma principale e questa viene trovata. Il problema è che se in questo nuovo programma quella stessa variabile serve per altri scopi, non appena si invoca la procedura quest'ultima modifica in modo inaspettato il suo valore interferendo con il resto del programma.

Ne discende un'altra regola: tutte le variabili di lavoro di un sottoprogramma dovrebbero essere definite all'interno di quest'ultimo. Eventuali valori/variabili esterne verranno comunicati tramite un parametro.

Passaggio di parametri per indirizzo (by reference)

Come abbiamo visto, il passaggio di parametri per valore non consente la modifica di variabili esterne. Qualche volta però questo è proprio ciò di cui abbiamo bisogno... Immaginiamo di volere scrivere un sottoprogramma per il calcolo delle due radici reali (se esistono) di un'equazione di secondo grado:

```
function equazione2grado(a, b, c: real) :  real;
begin
    .... Calcoli ...
end;
```

Gli unici valori di cui questa funzione ha bisogno sono i coefficienti del termine di secondo grado, del termine di primo grado e del termine noto (per applicare la notissima formula $x_{12} = \frac{-b \mu \sqrt{b^2 - 4ac}}{2a}$). Ecco quindi giustificata la presenza dei tre parametri chiamati a, b e c. Purtroppo, però, una funzione può restituire <u>un solo valore</u>: impossibile, quindi, far restituire con i meccanismi noti <u>i due</u> valori corrispondenti alle due soluzioni richieste. La situazione parrebbe senza vie d'uscita: o si fa restituire la funzione la prima delle due radici (X1) oppure si fa restituire la seconda (X2).

Qualcuno potrebbe essere tentato dalla seguente strada (ma, come appena visto, <u>assolutamente da evitare)</u>:

```
program equazioni;
var x1,x2: real;

function equazione2grado(a, b, c: real) :  real;
begin
    .... Calcoli ...
    x1:= ...;  x2:= ...;

end;
```

Questa soluzione utilizza direttamente due variabili del programma principale. Si tratta di una pratica assolutamente da evitare per tutti i motivi visti in precedenza.

Quello che occorre è un modo 'sicuro' per consentire ad un sottoprogramma la modifica di una variabile esterna; sicuro significa che un programmatore che sta per utilizzare un sottoprogramma è in grado, consultando la documentazione, di capire che la variabile che sta indicando come parametro attuale può essere modificata dal sottoprogramma ed in che modo.

Tutto questo accade con l'utilizzo dei **parametri passati per indirizzo (by reference)**. A livello sintattico la modifica è minima: basta aggiungere prima del nome di un parametro formale la parola var:

```
program equazioni;
var soluzione1, soluzione2: real;

procedure equazione2grado(a, b, c: real; var x1,x2: real);
begin
    .... Calcoli ...
    x1:= ...;  x2:= ...;

end;

begin
   equazione2grado(3,4,5, soluzione1, soluzione2);
   writeln('Le soluzione sono ', soluzione1, soluzione2);
    readln;
end.
```

Quando un parametro formale è specificato per indirizzo una modifica ad esso si ripercuote in modo permanente sulla variabile utilizzata come parametro attuale al momento della chiamata.

Detto in altre parole: nella sezione delle variabili del programma principale sono state dichiarate due variabili (soluzione1 e soluzione2) per memorizzare i risultati che verranno calcolati dalla procedura.

La procedura viene poi richiamata indicando proprio queste due variabili come ultimi due parametri attuali.

Poiché soluzione1 corrisponde al parametro formale X1 e poiché X1 è stato dichiarato per indirizzo, ogni modifica fatta ad X1 si ripercuoterà anche su soluzione1;

ESERCIZI SVOLTI – primo blocco

SOT 1. difficoltà: bassa Che percentuale rappresenta un numero A rispetto ad un altro numero B? Esempio: 100 rispetto a 150 = 75 (100 è il 75% di 150).

```
program percentuale;
uses newdelay,crt;

(* che percentuale è A di B? *)
function perc(A, B: real): real;
begin
  perc := (A/B) * 100
end;

begin
  clrscr;

(* esempi d'uso ... *)
  writeln( perc(1, 1):3:2, '%' );  (* 100% *)
  writeln( perc(17, 34):3:2, '%' );  (* 50% *)
  writeln( perc(8, 24):3:2, '%' );  (* 33.33333333%  ...*)
  writeln( perc(234.78, 4356.87):3:2, '%' );  (* e chi lo sa ?? *)

(* viene calcolato un valore corretto anche quando A < B *)
  writeln( perc(150, 100):3:2, '%' );  (* 150% *)

  writeln('INVIO per continuare ...');
  readln;
end.
```

SOT 2. difficoltà: bassa Calcolare una certa percentuale di un numero.

```
program provaPercentuale;
uses newdelay, crt;

function percentuale(perc: real; numero: real): real;
begin
  percentuale := (numero/100) * perc;
end;

begin
  clrscr;
  writeln( percentuale(20, 100):2:2); (* 20% di 100 = 20 *)
  writeln( percentuale(20, 35):2:2); (* 20% di 35 = 7 *)
  writeln( percentuale(0, 47):2:2); (* 0% di qualunque numero = 0 *)
  writeln( percentuale(20, 0):2:2); (* 20% di 0 = 0 *)


  readln;

end.
```

SOT 3. difficoltà: bassa Calcolare un prezzo comprensivo d'IVA.

```
program provaPercentuale;
uses newdelay, crt;
var prezzo, iva: real;

(* sfruttiamo la funzione scritta prima ... *)
function percentuale(perc: real; numero: real): real;
begin
  percentuale := (numero/100) * perc;
end;

function prezzoConIVA(prezzoSenzaIVA: real; percentualeIVA: real): real;
begin
  prezzoConIVA := prezzoSenzaIVA + percentuale(prezzoSenzaIVA, percentualeIVA);
end;

begin
  clrscr;
  writeln('Inserire prezzo senza IVA');
  readln(prezzo);

  writeln('Inserire pecentuale IVA da applicare');
  readln(iva);

  writeln('Ecco il totale IVA compresa: ', prezzoConIVA(prezzo, iva):6:2);
  readln;

end.
```

SOT 4. difficoltà: bassa Conversione da metri a chilometri e viceversa.

```
program conversioni;
uses newdelay, crt;
var metri, km: real;

(* da m a km *)
function m_km(quantiMetri: real): real;
begin
  m_km := quantiMetri / 1000;
end;

(* da km a m *)
function km_m(quantiKm: real): real;
begin
  km_m := quantiKm * 1000;
end;

begin
  clrscr;
  writeln('Inserire metri');
  readln(metri);
  writeln(metri:5:2,' metri corrispondono a ',m_km(metri):5:2,' chilometri');

  writeln('Inserire ora i chilometri');
  readln(km);
  writeln(km:5:2,' chilometri corrispondono a ',km_m(km):5:2,' metri');
  readln;

end.
```

SOT 5. difficoltà: bassa Convertire un certo numero di secondi nei minuti corrispondenti. Si vogliono anche sapere i secondi che avanzano.

```
program conversioni;
uses newdelay, crt;
var ss, avanzano: integer;

(* da secondi a minuti; viene calcolato anche l'avanzo *)
function ss_mm(quantiSecondi: integer; var avanzo: integer): integer;
begin
  avanzo := quantiSecondi mod 60;
  ss_mm := quantiSecondi div 60;
end;

begin
  clrscr;
  writeln('Inserire secondi');
  readln(ss);

  writeln(ss,' secondi corrispondono a ',ss_mm(ss,avanzano),' minuti e ', avanzano, ' secondi');

  readln;

end.
```

SOT 6. difficoltà: bassa Convertire un certo numero di minuti nelle ore corrispondenti. Si vogliono anche sapere i minuti che avanzano.

```
program conversioni;
uses newdelay, crt;
var mm, avanzano: integer;

(* da secondi a minuti; viene calcolato anche l'avanzo *)
function ss_mm(quantiSecondi: integer; var avanzo: integer): integer;
begin
  avanzo := quantiSecondi mod 60;
  ss_mm := quantiSecondi div 60;
end;

(* da minuti a ore; viene calcolato anche l'avanzo *)
function mm_hh(quantiMinuti: integer; var avanzo: integer): integer;
begin
  (* sfrutto la precedente: il calcolo da fare e' infatti lo stesso ! *)
  mm_hh := ss_mm(quantiMinuti, avanzo);
end;

begin
  clrscr;
  writeln('Inserire minuti');
  readln(mm);

  writeln(mm,' minuti corrispondono a ',mm_hh(mm,avanzano),' ore e ', avanzano, ' minuti');
  readln;
end.
```

SOT 7. difficoltà: bassa Convertire un certo numero di secondi nelle ore corrispondenti. Si vogliono anche sapere i minuti ed i secondi che che avanzano.

```
program conversioni;
uses newdelay, crt;
var ss, avanzano, avanzano_mm: integer;

(* da secondi a minuti; viene calcolato anche l'avanzo *)
function ss_mm(quantiSecondi: integer; var avanzo: integer): integer;
begin
  avanzo := quantiSecondi mod 60;
  ss_mm := quantiSecondi div 60;
end;

(* da minuti a ore; viene calcolato anche l'avanzo *)
function mm_hh(quantiMinuti: integer; var avanzo: integer): integer;
begin
  (* sfrutto la precedente: il calcolo da fare e' infatti lo stesso ! *)
  mm_hh := ss_mm(quantiMinuti, avanzo);
end;

(* da secondi a ore; viene calcolato anche l'avanzo *)
function ss_hh(quantiSecondi: integer; var avanzo_ss, avanzo_mm: integer): integer;
var quantiMinuti: integer;
begin
  quantiMinuti := ss_mm(quantiSecondi, avanzo_ss);
  ss_hh := mm_hh(quantiMinuti, avanzo_mm)
end;

begin
  clrscr;
  writeln('Inserire i secondi per convertirli in ore ...');    readln(ss);

  write(ss,' secondi corrispondono a ',ss_hh(ss,avanzano, avanzano_mm),' ore, ');
  writeln(avanzano_mm, ' minuti e ', avanzano,' secondi');
  readln;
end.
```

SOT 8. difficoltà: bassa Estrarre la parte decimale di un numero reale X (quella 'dopo la virgola'); in pratica simula la funzione standard frac... Esempio: decimale(13,75) - > 0,75.

```
program parteDecimale;
uses newdelay,crt;
(*   NOTA. Con il Pascal la parte decimale e' separata da quella intera da un punto, non dalla virgola;
   NOTA. Viene sfruttata la funzione predefinita TRUNC (che elimina la parte decimale da un reale.
            TRUNC(10.7) -> 10  *)

function decimale(x: real): real;
begin
  decimale := x - trunc(x);
end;

begin
  clrscr;

(* esempi d'uso ... *)
  writeln( decimale(0.8917):4:4 );  (* 0.8917 *)
  writeln( decimale(4.8917):4:4 );  (* 0.8917 *)
  writeln( decimale(368.8917):4:4 );  (* 0.8917 *)

  writeln('INVIO per continuare ...');     readln;   end.
```

SOT 9. difficoltà: bassa Dato un carattere dire se rappresenta una lettera maiuscola.

```
program controllaMaiuscolo;
uses newdelay,crt;

(* restituisce true solo se il carattere e' una lettera maiuscola cioe' solo se il suo codice ascii e' compreso tra quello
della A e quello della Z *)
function isUpCase(c: char) : boolean;
begin
  isUpCase := ( ord(c) >= ord('A') ) and ( ord(c) <= ord('Z') );
end;

begin
  clrscr;

(* esempi d'uso ... *)
  writeln( isUpCase('A') );
  writeln( isUpCase('a') );
  writeln( isUpCase('Z') );
  writeln( isUpCase('z') );
  writeln( isUpCase('!') );

  writeln('INVIO per continuare ...');
  readln;

end.
```

SOT 10. difficoltà: media Dato un carattere trasformarlo in minuscolo; se è già minuscolo lasciarlo tale. Sfruttare la funzione precedente (vedi SOT9)

```
program convertiInMinuscolo;
uses newdelay,crt;

(* restituisce true solo se il carattere e' una lettera maiuscola cioe' solo se il suo codice ascii e' compreso tra quello
della A e quello della Z *)
function isUpCase(c: char) : boolean;
begin
  isUpCase := ( ord(c) >= ord('A') ) and ( ord(c) <= ord('Z') );
end;

(* Converte un SINGOLO carattere in minuscolo; se non ha senso la conversione (non e' una lettera maiuscola)
restituisce il carattere inalterato*)
function lowCase(c: char) : char;
var distanza: integer;
begin

  if isUpCase(c) then
  begin
    (* che valore separa le lettere minuscole dalle maiuscole nel codice ascii? *)
    distanza := ord('a') - ord('A'); (* 97 - 65 = 32 *)

(* NOTA: e' meglio usare ord('A') invece del valore 65 perche' 65 funziona solo con la tabella dei caratteri dei sistemi
operativi Microsoft e altri, ma non tutti: ci sono sistemi operativi con tabelle con valori diversi; ord('A') ha piu'
probabilita' di funzionare: e' indipendente dal valore nella tabella (potrebbe essere 65 o 34 o qualsiasi altro valore,
ma se le lettere sono codificate sequenzialmente dalla 'A' alla 'Z' e dalla 'a' alla 'z', non ci saranno problemi: cambiera'
solo la distanza che rimarra' pero' fissa per quel sistema operativo; per gli stessi motivi e' meglio usare la formula
ord('a') - ord('A')  invece del suo valore 32 (valido solo in ambienti windows ...)
*)
```

Una sottoprogramma può sfruttarne un altro !

```
(* trasformo in minuscolo: al codice ascii del carattere sommo la sfasatura con il set delle minuscole, e ritrasformo il risultato nel carattere corrispondente *)
    lowCase := chr( ord(c) + distanza )
  end
  else
    lowCase:=c
end;


begin
  clrscr;

(* esempi d'uso ... *)
  writeln( lowCase('A') );
  writeln( lowCase('a') );
  writeln( lowCase('Z') );
  writeln( lowCase('z') );
  writeln( lowCase('!') );  (* inalterato ... *)

  writeln('INVIO per continuare ...');
  readln;

end.
```

SOT 11. difficoltà: alta Data una stringa convertirla in minuscolo o maiuscolo a seconda del valore di un parametro come (se come='m' converte in minuscolo, se come='M' converte in maiuscolo; se come non ha un valore valido la stringa viene restituita immodificata). Sfruttare alcune funzioni precedenti (SOT9 e SOT10)

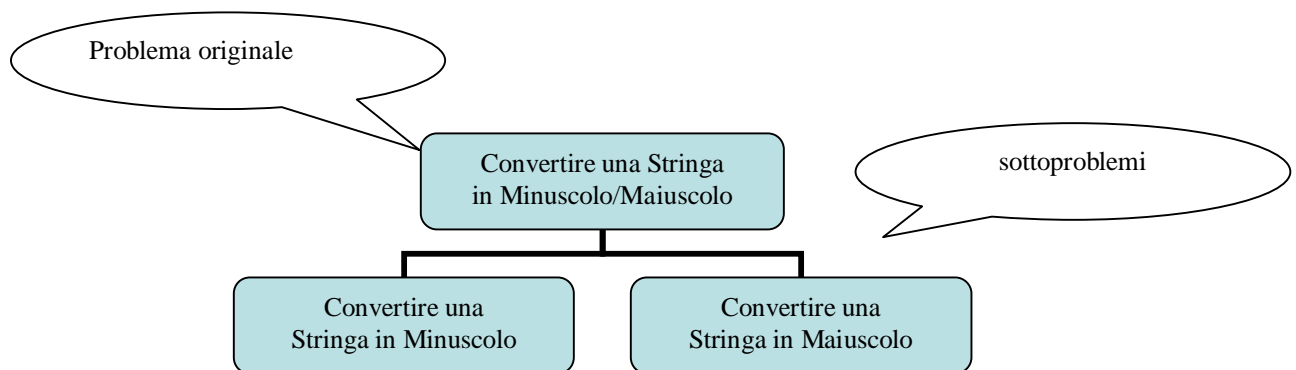
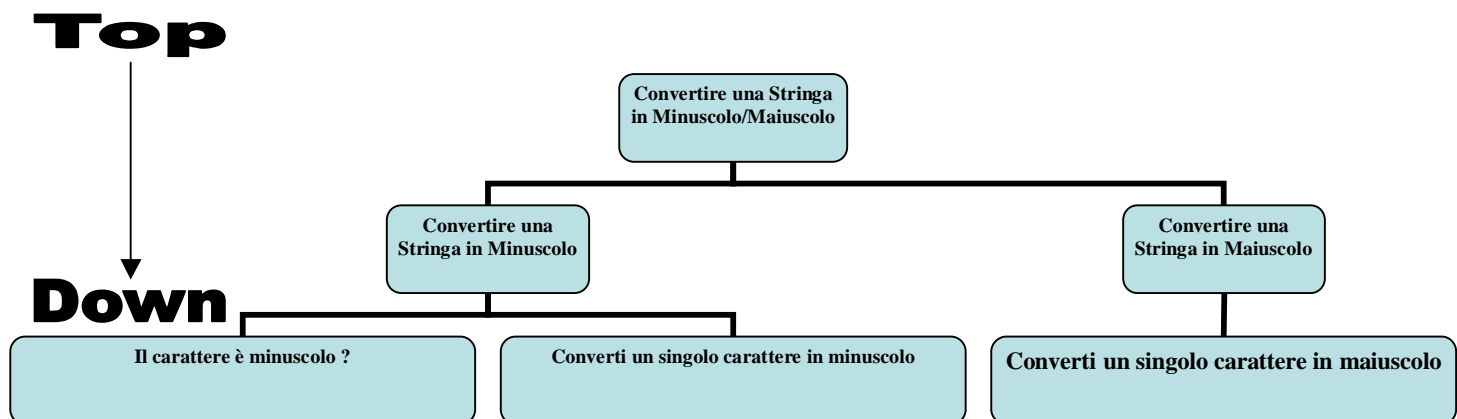
```
program MinuscoloMaiuscolo;
uses newdelay, crt;

(* per un commento vedi SOT9 *)
function isUpCase(c: char) : boolean;
begin  isUpCase := ( ord(c) >= ord('A') ) and ( ord(c) <= ord('Z') ); end;

(*per un commento vedi SOT10 *)
function lowCase(c: char) : char;
var distanza: integer;
begin
  if isUpCase(c) then
  begin
    distanza := ord('a') - ord('A'); (* 97 - 65 = 32 *)
    lowCase := chr( ord(c) + distanza )
  end
  else
    lowCase:=c
end;

(* converte una stringa in minuscolo *)
function minuscolo(s: string): string;
var i: integer; ris: string;
begin
  ris:='';
  for i:=1 to length(s) do
      ris := ris + lowCase(s[i]);

  minuscolo:=ris;

end;


(* Converte una stringa in maiuscolo *)
function maiuscolo(s: string): string;
var i: integer; ris: string;
begin
```

```
  ris:='';
  for i:=1 to length(s) do
    ris:=ris + upcase(s[i]);  (* sfrutta la funzione predefinita upcase ... *)

  maiuscolo:=ris;
end;

(* converte un  maiuscolo a seconda del parametro 'come'; se come='m'  converte in minuscolo, se come='M'
converte in maiuscolo; se come non  ha un valore valido la stringa viene restituita identica *)
function MiniMaiu(s: string; come: char) : string;
begin
  case come of
    'm': MiniMaiu := minuscolo(s);
    'M': MiniMaiu := maiuscolo(s);
  else
    MiniMaiu := s;
  end;
end;


begin
  clrscr;

(* esempi d'uso ... *)
  writeln( MiniMaiu('AaZz12!','m') );
  writeln( MiniMaiu('AaZz12!','M') );
  writeln( MiniMaiu('AaZz12!','t') );


  writeln('INVIO per continuare ...');
  readln;

end.
```

Sfruttando i sottoprogrammi già esistenti, la soluzione del problema originale diventa banale.

Metodologie Top Down e Bottom Up

L'esercizio precedente introduce in modo naturale una tecnica di sviluppo utilizzata per problemi di media/alta complessità. Invece di tentare di risolvere il problema come un tutt'uno, compito di solito molto difficile, si procede ad una suddivisione in sottoproblemi. E' intuitivo che ogni sottoproblema sia più semplice di quello originale. Per ogni sottoproblema che si ritiene ancora troppo complesso si procede ad ulteriore suddivisione. Il procedimento viene ripetuto fino ad ottenere un certo numero di sottoproblemi sufficientemente semplici da essere risolti e codificati con una certa facilità. A livello pratico questo coincide ad individuare tutta una serie di sottoprogrammi che a loro volta ne richiamano altri. Questa metodologia viene chiamata TOP DOWN (dall'alto al basso), perché graficamente possiamo rappresentarla come una struttura a piramide in cui in cima (top) si mette il problema originale e via via che si scende di livello sottoproblemi sempre più semplici, fino ad arrivare alla base (bottom) della piramide in cui troviamo i sottoproblemi più semplici.

Vediamo di riconoscere questo procedimento nell'esercizio precedente. Si trattava di realizzare un comando per trasformare una stringa in minuscolo o in maiuscolo. Questo problema iniziale viene suddiviso in modo molto naturale nei due sottoproblemi 1) convertire una stringa in minuscolo e 2) convertire una stringa in maiuscolo:

Il sottoproblema 'convertire una stringa in minuscolo' può essere ulteriormente scomposto in due altri sotto problemi: 1) decidere se un carattere è minuscolo e 2) convertire un singolo carattere in minuscolo (più semplice rispetto alla conversione di un'intera stringa). Anche per l'altro sotto problema viene individuato il sotto problema corrispondente alla conversione in maiuscolo di un singolo carattere.

NOTA: questa tecnica viene indicata anche come '*divide et impera*' (dividi e mantieni sotto controllo) da un detto degli antichi romani: per mantenere sotto controllo un vasto territorio appena conquistato vaste fette della popolazione venivano deportate in aree geografiche lontane. In questo modo quel popolo perdeva la sua unità e la sua forza ed era più facile dominarlo.

Oltre alla riduzione della complessità la metodologia top-down offre un'altra caratteristica: consente di allestire la struttura portante dell'intero programma senza aver completato i singoli componenti, così da poterlo provare nei suoi aspetti macroscopici. Per fare un paragone è come se avessimo a disposizione il prototipo di un'automobile con il telaio (ma non i vetri e le rifiniture), il volante e le ruote (ma un finto servosterzo e niente copertoni), il cruscotto con i controlli ma senza un vero impianto elettrico, pedali e cambio ma niente sistema di frenatura e solo un abbozzo del motore. Certamente tutto questo non ci permetterebbe di fare un giro di pista a Monza ma consentirebbe agli ingegneri ed ai meccanici di salire veramente in macchina e saggiare dal vivo almeno le caratteristiche e economiche (visibilità, comodità del posto guida, spazi interni, corretta disposizione dei controlli eccetera); insomma non è come avere singoli pezzi, pur funzionanti ma che non consentono di trovare il tutto almeno in alcuni dei suoi aspetti.

Di nuovo, cerchiamo di ritrovare questi concetti nell'esempio precedente. Dopo aver individuato i sottoprogrammi da sviluppare il programmatore può rapidamente scrivere solo l'intelaiatura di ciascuno di essi ma già collegando il tutto a formare un programma funzionante:

```
program MinuscoloMaiuscolo;
uses newdelay, crt;

function isUpCase(c: char) : boolean;
begin
   isUpCase := true;
end;

function lowCase(c: char) : char;
begin
    lowCase := 'a';
end;

function minuscolo(s: string): string;
begin
  minuscolo:='stringa di prova';
end;

function maiuscolo(s: string): string;
  maiuscolo:='STRINGA DI PROVA';
end;

function MiniMaiu(s: string; come: char) :
string;
begin
  case come of
    'm': MiniMaiu := minuscolo(s);
    'M': MiniMaiu := maiuscolo(s);
  else
    MiniMaiu := s;
  end;
end;

begin
  clrscr;

(* esempi d'uso ... *)
  writeln( MiniMaiu('AaZz12!','m') );
  writeln( MiniMaiu('AaZz12!','M') );
  writeln( MiniMaiu('AaZz12!','t') );

  writeln('INVIO per continuare ...');
  readln;

end.
```

Ecco il ‘trucco’: pur avendo stabilito l'interfaccia di ogni sottoprogramma (il nome ed i parametri che riceverà) il codice di ciascuno di essi è ridotto all'osso, il minimo indispensabile per poterli richiamare i poter provare il programma principale.

Ogni funzione non calcola veramente il suo risultato ma restituisce un valore fasullo (però del tipo giusto): ad esempio la funzione maiuscolo restituisce come risultato la stringa fissa ‘ STRINGA DI PROVA’.

Questo permette comunque a programmatore di scrivere nei minimi dettagli la funzione *MiniMaiu* che, per così dire, non si accorge di richiamare delle funzioni incomplete (anche se, naturalmente, i valori che si vede restituire sono sempre quelli e fasulli.

Il programmatore può anche organizzare il corpo principale del programma verificando che tutti i sottoprogrammi si richiamino correttamente tra loro. Fatto questo può procedere a completare veramente ciascun sottoprogramma oppure, e questo è un altro vantaggio dell'aver costruito rapidamente lo ‘scheletro’ del programma, può fidare il completamento a più programmatori in contemporanea e da ciascuno consegnare una copia di questo scheletro in modo da poter testare il codice senza avere a disposizione la parte che viene sviluppata dagli altri.

Il completamento può avvenire anche per affinamenti successivi: si iniziano con l'introdurre versioni non perfette ma via via sempre più funzionanti dei vari sottoprogrammi. Questo aiuta a concentrarsi sugli aspetti fondamentali del problema: ad esempio non è importante all'inizio preoccuparsi di tutti i possibili errori di inserimento dati che potrebbe commettere l'utente; la gestione degli errori è senz'altro un aspetto che può essere introdotto in un secondo momento.

Naturalmente dopo che ciascun programmatore avrà completato il test sulla sua parte sarà necessaria una fase di test conclusiva con tutto il codice al suo posto!

La tecnica del top-down presuppone, però, che il programmatore abbia un'idea chiara, almeno a grandi linee, del programma nella sua versione completa. Solo in questo modo potrà infatti prefigurarsi i sottoprogrammi che dovranno essere sviluppati e come questi dovranno interagire tra loro. Diversamente non sarebbe infatti in grado di abbozzare lo scheletro dell'intero programma.

Bottom-Up
Quando questo non è possibile (per l'estrema complessità del progetto e/o per l'alto grado di innovazione dello stesso che non consente di avere le idee chiare su tutto fin dall'inizio) può essere d'aiuto procedere in un altro modo: cominciare a sviluppare piccole porzioni sulle quali si hanno comunque idee chiare senza sapere ancora come si incastreranno tra loro (almeno nei dettagli). Sarà solo in un secondo momento che ci si preoccuperà di assemblare il programma con questi componenti. Per fare un paragone, è un po' come se si volesse realizzare un veicolo innovativo per spostarsi sulla sabbia: il progettista potrebbe non aver ancora le idee chiare sulla forma e le caratteristiche definitive del veicolo. Potrebbe però decidere di cominciare a realizzare delle ruote speciali, per poi passare al motore, poi alle singole parti del telaio e così via. Infine deciderà come assemblare insieme queste parti: certamente potrebbe essere costretto a rifare dei pezzi o a scartarne alcuni o ad adattarne degli altri (è il prezzo per questa modalità di sviluppo in cui non è possibile progettare nei dettagli fin dall'inizio); inoltre, di nuovo non avendo progettato fin dall'inizio le interconnessioni tra le varie parti, sarà più difficile affidarne lo sviluppo a diversi programmatori senza essere costretti a dell'ulteriore lavoro in fase di assemblaggio (un programmatore potrebbe aver progettato l'interfaccia di un suo sottoprogramma in modo non ottimale per un altro).

Questa tecnica è chiamata BOTTOM-UP (dal basso verso l'alto): si parte dai dettagli e dai componenti più semplici per assemblare via via i componenti di più alto livello, quelli più complessi.

Non si può dire che la tecnica sia superiore all'altra, tant'è che non è infrequente il caso in cui vengano in realtà utilizzate entrambe: il progetto viene fatto magari in modo top-down ma lo sviluppo parte in modo dettagliato dei livelli più bassi. La tecnica bottom-up si rivela particolarmente efficace quando un progetto viene sviluppato seguendo la filosofia della OOP (Object Oriented Programmino) che vede programmatore sviluppare componenti autonomi con l'obiettivo di poterli facilmente riutilizzare in progetti diversi. L'OOP verrà affrontata in quarta.

ESERCIZI SVOLTI – secondo blocco

SOT 12. difficoltà: bassa Dati due numeri determinare il maggiore.

```
program massimo;

var n1,n2,x: real;

(* restituisce il massimo tra due valori comunicati *)
function max(a,b: real): real;
var risultato: real;
begin
  if a>=b then
    risultato:=a
  else
     risultato:=b;
   max:=risultato
end;

(* prove di utilizzo della funzione massimo *)
begin
  (* specificando come parametri delle costanti numeriche ... *)
  writeln ( max(3,46):2:0 );
  writeln ( max(46,3):2:0 );
  writeln ( max(3,3):2:0 );
  writeln ( max(-43,1) :2:0);

  (* specificando come parametri delle variabili esterne *)
  writeln('Inserisci due numeri e ti diro'' qual''e'' il piu'' grande');
  write('Dimmi il primo -> '); readln(n1);
  write('Dimmi il secondo -> '); readln(n2);

  writeln ( max(n1,n2) :2:0);
  writeln ( max(n1,46) :2:0);
  writeln ( max(46,n2) :2:0);

  (* una funzione puo' essere usata direttamente nel calcolo di un'espressione *)
  x:=13 * ( max(n1,6) - 18);
  writeln('X: ',x:2:0);

  (* max puo' essere usata nella condizione di un if *)
  if max(n1,n2) > 100 then
    writeln('Il massimo tra i due valori supera 100');

  (* o di un repeat ... *)
  (* ad una centralina di controllo arrivano i dati sull'inquinamento letti da due sonde poste in punti strategici della
citta'; leggere questi dati fino a quando una delle due sonde comunica un valore maggiore di 50 (microgrammi/mc)
*)

  repeat
    write('Inserire valore sonda n. 1 -> ');
    readln(n1);

    write('Inserire valore sonda n. 2 -> ');
    readln(n2)
  until max(n1,n2)>50; (* grazie alla funzione max il controllo può essere effettuato in contemporanea *)

  readln;
end.
```

SOT 13. difficoltà: bassa Determinare il numero di vocali presenti in una stringa.

```
program prova;
uses newdelay,crt;

(* conta quante vocali ci sono in una stringa, prima tecnica *)
function contaVocali(s: string) : integer;
var nv,i: integer;

begin
  nv:=0;  (* numero vocali *)
  for i:=1 to length(s) do
    case s[i] of
      'a','A','e','E','i','I','o','O','u','U': nv:=nv+1;
    end;

  contaVocali:=nv
end;

(* conta quante vocali ci sono in una stringa, secondo tecnica *)
(* usa la funzione POS invece del case *)
function contaVocali2(s: string) : integer;
var nv,i: integer;

begin
  nv:=0;  (* numero vocali *)
  for i:=1 to length(s) do
    if pos(s[i], 'aAeEiIoOuU')<>0 then
      nv:=nv+1;
  contaVocali2:=nv
end;

begin
  writeln(contaVocali('casa dolce casa'));
  readln
end.
```

SOT 14. difficoltà: bassa Scrivere un comando che invita l'operatore a premere un tasto per continuare.

```
(* ha bisogno di commenti ? *)
procedure attendi;
begin
  writeln('............. PREMI INVIO PER CONTINUARE ..............');
  readln
end;
```

SOT 15. difficoltà: bassa Scrivere un comando per la visualizzazione di un messaggio qualsiasi tra due cornicette di asterischi; l'esecuzione del programma deve anche interrompersi invitando l'utente a per un tasto per continuare (sfruttare il comando precedente).

```
Program prova;
procedure attendi;
begin
  writeln('............. PREMI INVIO PER CONTINUARE ..............');
  readln
end;

(* visualizza la stringa ricevuta ed attende la pressione di INVIO *)
procedure messaggio(mes: string);
begin
  writeln('--------------------------------------------');
  writeln(mes);
  writeln('--------------------------------------------');
  attendi;
end;

begin
  clrscr;
  messaggio('Ciao, come va?');
end.
```

SOT 16. difficoltà: bassa Scrivere un sottoprogramma per il calcolo di x^y con X e Y interi positivi.

```
program potenze;
uses newdelay, crt;
var unaBase,unEsponente: integer;

function eleva(base,esponente: integer):real;
var risultato: real; i: integer;
begin
  risultato:=1;

  for i:=1 to esponente do
    risultato:=risultato*base;

  eleva:=risultato
end;

begin
  clrscr;
  write('Dimmi la base '); readln(unaBase);
  write('Dimmi l''esponente '); readln(unEsponente);

  writeln('base: ',unaBase,' - esponente: ',unEsponente);
  writeln('il risultato e'': ', eleva(unaBase,unEsponente):6:0 );
  readln
end.
```

SOT 17. difficoltà: media Togliere da una stringa eventuali spazi inutili all'inizio o alla fine della stringa stessa.

```
program elimina_spazi;
uses crt;

var stringa: string;

function togli_spazi(s: string): string;
var i: integer;
    inizio,fine: integer;
    spazio: boolean;
    risultato: string;
begin
  inizio:=1; fine:=length(s);

  (* cerco il primo carattere diverso da spazio a sinistra *)
  spazio:= (s[inizio]=' ');
  while ( inizio<=fine ) and spazio do
  begin
    inc(inizio);
    spazio:=(s[inizio]=' ')
  end;

  (* cerco il primo carattere diverso da spazio a destra *)
  spazio:= (s[fine]=' ');
  while ( fine>=inizio ) and spazio do
  begin
    dec(fine);
    spazio:=(s[fine]=' ')
  end;

  risultato:='';
  for i:=inizio to fine do
    risultato:=risultato+s[i];

  togli_spazi:=risultato
end;

begin
  clrscr;
  writeln( '#',togli_spazi(''),'#' );
  writeln( '#',togli_spazi('a'),'#' );
  writeln( '#',togli_spazi(' a'),'#' );
  writeln( '#',togli_spazi('a '),'#' );
  writeln( '#',togli_spazi('ab'),'#' );
  writeln( '#',togli_spazi('  a'),'#' );
  writeln( '#',togli_spazi('a  '),'#' );
  writeln( '#',togli_spazi('  ab'),'#' );
  writeln( '#',togli_spazi('ab  '),'#' );
  writeln( '#',togli_spazi('  ab   '),'#');
  writeln( '#',togli_spazi('  questa e" una frase '),'#' );
  readln
end.
```

REGOLE DI VISIBILITA' (SCOPING RULES) E DURATA DELLE VARIABILI

Scrivendo programmi di una certa complessità è importante capire per ogni entità (variabili, costanti, sottoprogrammi, tipi ecc.) dove può essere usata e la sua durata (che non è detto che si estenda dal momento in cui il programma viene fatto partire a quello in cui viene terminato).

Quando un'applicazione è costituita da un unico file sorgente e non si fa uso di sottoprogrammi il problema non sussiste: in un qualsiasi punto del codice tra il *begin* di inizio e l'*end* finale possiamo far riferimento ad una variabile qualsiasi, ad una costante qualsiasi ecc; queste stesse variabili esistono dal momento in cui il programma inizia la sua esecuzione fino al momento in cui il programma viene terminato.

La situazione si complica quando in un programma si dichiarano dei sottoprogrammi (procedure o funzioni): infatti possiamo avere parametri con lo stesso nome di variabili o costanti del programma principale, oppure variabili o costanti dichiarate nelle rispettive sezioni del sottoprogramma e che hanno lo stesso nome di variabili e costanti del programma principale. Cerchiamo di mettere a fuoco la situazione con un esempio:

```
program principale;

var A: integer;  S: string;
     X: real;

procedure P(A: integer);
var S: real;
begin
  write(A);
  S := 3.14*X;
end;

begin
  A := 4;
  S := 'ciao';
end.
```

In quali punti del codice è visibile la variabile A dichiarata nella sezione *var* del programma principale?

Quando ci troviamo all'interno del blocco *begin … end.* del programma principale e tentiamo di assegnare un valore alla variabile A, viene usata quella dichiarata nella sezione *var* del programma principale oppure quella dichiarata come parametro per la procedura P?

E quando ci troviamo all'interno del blocco *begin … end.* del programma principale e tentiamo di assegnare un valore alla variabile S, viene usata quella dichiarata nella sezione *var* del programma principale oppure quella dichiarata come variabile locale nella procedura P?

La variabile A usata con l'istruzione *write* nella procedura è il suo parametro o la variabile A dichiarata nella sezione *var* del programma principale?

La variabile S usata con l'assegnamento nella procedura è la sua variabile locale o, di nuovo, quella della sezione *var* del programma principale?

La variabile A usata nel corpo principale del programma è quella dichiarata nella sezione *var* del programma principale o il parametro della procedura?

E infine: la variabile S usata nel corpo principale del programma è quella dichiarata nella sezione *var* del programma principale o la variabile locale con lo stesso nome dichiarata nella procedura?

Come potete ben capire è essenziale stabilire delle regole, pena il caos totale!

- Le entità dichiarate nella sezione *var* del programma principale sono visibili:
  - in un punto qualsiasi tra il *begin* e l'*end.* del programma principale
  - in un sottoprogramma qualsiasi a patto che nessun parametro o variabile locale abbiano lo stesso nome: in questo ultimo caso tra il *begin* e *l'end* hanno il sopravvento i parametri delle variabili locali (potremmo dire che si fa sempre riferimento alla dichiarazione 'più vicina' al punto in cui stiamo scrivendo);

    NOTA: abbiamo comunque più volte ribadito che un sottoprogramma non dovrebbe mai tentare di utilizzare direttamente una variabile esterna (sempre e solo attraverso un parametro!)
- Le entità dichiarate nella sezione *var* del programma principale esistono dal momento in cui inizia il programma fino al momento in cui termina.

Le entità dichiarate a livello di sottoprogramma (parametri e variabili locali) sono visibili:

- tra il *begin* e *l'end* di quel sottoprogramma; parametri e variabili locali non possono quindi nessun modo essere utilizzati all'esterno del sottoprogramma in cui sono dichiarati;
- in altri sottoprogrammi ma solo se dichiarati all'interno del primo (e a patto che questi ultimi a loro volta non utilizzino parametri o loro variabili locali con lo stesso nome);

- Le entità dichiarate a livello di sottoprogramma (parametri e variabili locali) vengono create nel momenti in cui inizia l'esecuzione del sottoprogramma e vengono distrutte nel momenti in cui l'esecuzione del sottoprogramma termina; ne deriva che quando un sottoprogramma viene chiamato più volte i valori assunti in precedenza dalle variabili locali non sono più disponibili.

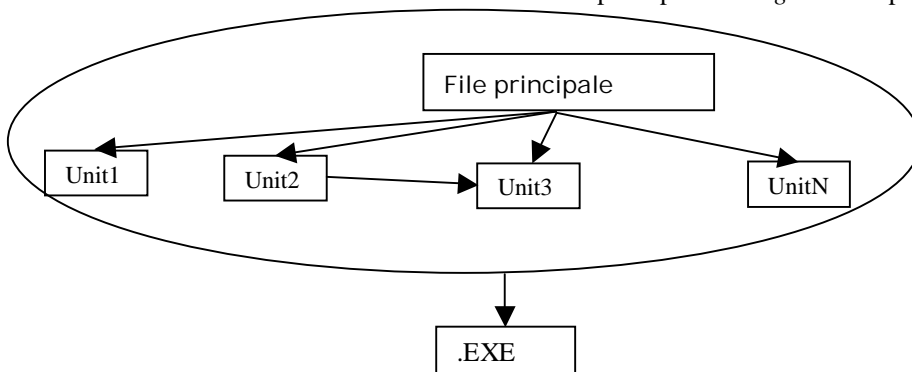
Applicando queste regole all'esempio precedente: la variabile A del programma può essere usata nel corpo principale ma non nella procedura (il parametro A la oscura); stessa situazione per la stringa S del programma (non può essere utilizzata nella procedura perché oscurata dalla variabile locale S; la variabile X può essere utilizzata invece dappertutto, nel programma e nella procedura (pratica, quest'ultima, da sconsigliare); il parametro A e la variabile locale S della procedura possono essere utilizzati solo in quest'ultima.

La letteratura informatica indica anche con la dicitura *ambiente locale* l'insieme delle entità dichiarate all'interno di una stessa entità sintattica (programma, sottoprogramma); ad esempio l'ambiente locale per una procedura è costituito dai suoi parametri e dalle sue variabili locali. Ciò che è invece dichiarato all'esterno di una certa unità sintattica viene invece indicato come *ambiente non locale*. Ad esempio, rispetto ad un sottoprogramma le variabili globali fanno parte dell'ambiente non locale. Attenzione però a non far corrispondere l'ambiente non locale con le variabili globali: se consideriamo infatti il caso di un sottoprogramma dichiarato all'interno di un altro per quest'ultimo l'ambiente non locale è costituito certamente delle variabili globali ma anche dalle variabili locali alla sottoprogramma che lo contiene:

```
program prova;
var x: real;
procedure P1;
var y: real;
  procedure P2;
  var z: real;
  begin
  end;
begin
end;
begin end.
```

La procedura P2 è dichiarata all'interno di P1. Il senso di ciò sta nel fatto che P2 svolge un compito molto particolare che a senso solo nell'ambito di P1. Dichiarata in questo modo, solo P1 può invocare P2 evitando usi maldestri.

L'ambiente non locale di P2 è costituito non solo dalla variabile x del programma principale ma anche dalla variabile y della procedura P1.

Quando l'applicazione è il risultato della compilazione e link di più sorgenti la situazione si complica: in generale bisogna decidere cosa concedere in utilizzo di un certo sorgente quando stiamo scrivendo il codice di un altro. I linguaggi si differenziano per possibilità e notazioni sintattiche anche molto diverse. Di solito è comunque possibile indicare un'entità dichiarata nella sezione *var* principale come *globale* rispetto agli altri file sorgenti; ad esempio, dare la possibilità dal file sorgente 'unit1' di usare una variabile globale della 'Unit2' o di richiamare un sottoprogramma di una qualsiasi altra Unit. In alcuni linguaggi, turbo Pascal compreso, per accedere all'entità di un altro file si deve rendere palese la volontà di usare quest'ultimo con una direttiva al compilatore (uses crt non vi dice nulla?).

TIPO DI DATO ARRAY

Voglio presentarvi alcune situazioni ‘problematiche’ per evidenziare l’inadeguatezza degli strumenti da programmatore che sono attualmente in vostro possesso.

Caso di studio 1: a proposito di medie …

Avete fatto carriera e vi ritrovate prof. Informatica! Avete anche appena terminato di correggere il primo pacco di compiti (sigh) e vorreste contare quanti alunni hanno preso un voto superiore alla media della classe: che informatici sareste se non pensaste di utilizzare un programmino per calcolare una volta per tutte questo valore ?

```
Program medie;

var
  i, voto: integer; (* ‘i’ serve per i cicli, ‘voto’ per leggere da tastiera il voto *)
  somma_voti, num_alunni: integer; (* num_alunni: numero degli alunni *)
  quanti_sopra_media: integer;
  media: real;

begin
  somma_voti:=0; media:=0; quanti_sopra_media:=0;

  writeln(‘Quanti alunni ci sono? ‘);
  readln(num_alu);

  for i:=1 to num_alu do
  begin
    writeln(‘Inserisci voto prossimo alunno: ‘);
    readln(voto);
    somma_voti:=somma_voti +  voto
  end;

  media:=somma_voti /num_alu;
```

Dopo aver impiegato solo pochi millesimi di secondo per scrivere questo codice, vi accorgete di … non sapere come andare avanti! Infatti ora che avete calcolato la media dovreste contare i voti che la superano, ma non avete più i voti a disposizione per confrontarli ! Infatti la variabile *voto* conterrà solo l’ultimo voto letto. I precedenti sono stati sommati nella variabile *somma_voti* e poi soprascritti ciascuno dal successivo (questo ovviamente perché la *readln* utilizza nel ciclo sempre la stessa variabile per leggere i voti).

è è Una prima orripilante soluzione (sicuri di esservi laureati in Informatica??) potrebbe essere quella di chiedere una seconda volta all'operatore gli stessi voti per contare quelli sopra la media

(si immagina di continuare il programma di prima):

```
for i:=1 to num_alu do
begin
  writeln('Inserisci voto prossimo alunno: ');
  readln(voto);

 if voto>media then
  quanti_sopra_media:= quanti_sopra_media + 1

end;
```

Stiamo scherzando? A parte le furiose proteste dell'utente che si vede costretto a ripetere l'inserimento dei dati, provate a pensare al rischio di inserire dati diversi dai precedenti con risultati del tutto falsati…

ATTENZIONE: la seconda 'soluzione' che viene proposta qui di seguito è, come il primo, solo un esempio. Come tale è NECESSARIO per apprezzare e capire l'uso del nuovo strumento di programmazione che presenterò, ma non deve essere 'studiato'.

è è Seconda soluzione (lascio decidere a voi se più o meno orripilante della precedente): 'banale, come ho fatto a non pensarci prima: basta usare una variabile diversa per ogni voto!'

```
Program medie;

var
  i, somma_voti, num_alunni, quanti_sopra_media: integer;
  media: real;
voto1, voto2, voto3, voto4, ..., voto32 (aiuto!): integer;

begin
  soma_voti:=0; media:=0; quanti_sopra_media:=0;

  writeln('Quanti alunni ci sono? ');
  readln(num_alu);
```

Il bello arriva adesso! Poiché non si sta usando una sola variabile (voto) per leggere i voti, non è possibile usare un ciclo per leggere i voti! E' necessario leggere il primo. Poi se il numero di alunni è maggiore di 1, significa che devo sicuramente leggerne anche un secondo; poi se il numero di alunni è anche maggiore di due significa che devo leggerne almeno anche un terzo e così via …

```
  writeln('Inserisci voto primo alunno: ');
   readln(voto1);
  somma_voti:= somma_voti + voto1

if num_alu>1 then
  begin
   writeln('Inserisci voto secondo alunno: ');
    readln(voto2);
   somma_voti:= somma_voti + voto2
  end;
```

```
if num_alu>2 then
begin
 writeln('Inserisci voto prossimo alunno: ');
  readln(voto3);
 somma_voti:= somma_voti + voto3
end;
```

e così via fino a ...

```
if num_alu>31 then
begin
 writeln('Inserisci voto 32-mo alunno: ');
  readln(voto32);
 somma_voti:= somma_voti + voto
end;

   media:=somma_voti / num_alu;  (* alleluia ... *)
```

e ora … ripetiamo tutto per contare quelli sopra la media

```
if voto1>media then
  quanti_sopra_media:= quanti_sopra_media + 1;

if num_alu>1 then
   if voto2>media then
    quanti_sopra_media:= quanti_sopra_media + 1;

  ecc. ecc. (abbiate pietà !)
```

Non so voi, ma io ho già perso di vista l'obiettivo che ci eravamo posti... Immaginate se invece degli alunni di una sola classe considerassimo quelli dell'intero istituto per una qualche statistica!

ED ECCO LA SOLUZIONE …

Quello che ci serve è un contenitore (si tratterà poi pur sempre di una variabile, anche se di tipo particolare) capace di contenere non un solo voto ma un insieme di voti. Il nome del contenitore in questo modo sarebbe uno solo, risolvendo l'inconveniente del proliferare selvaggio del numero delle variabili; inoltre, come vi spiegherò tra poco, sarà ancora possibile usare i cicli per elaborare con poche righe di codice tutti i dati.

VOTI

7	4	5	6	6	8	8	5
voti[1]	voti[2]	voti[3]	voti[4]	voti[5]	voti[6]	ecc.	

- Il nome del contenitore è VOTI. Questo è il nome che il programmatore ha deciso di dare all'*ARRAY*. Quando l'array è come quello nel disegno soprastante (lineare, monodimensionale) è anche chiamato *vettore*. Esistono *array* bidimensionali (pensate ad una griglia tipo battaglia navale, o ad uno schema tipo parole crociate) chiamati *matrici*.

- I dati non sono depositati nel vettore alla rinfusa ma tenuti in 'scomparti' separati.

- Ogni 'scomparto' occupa una precisa posizione (1, 2, 3, 4 ecc.) specificata tramite un numero detto *indice* che va indicato tra parentesi quadrate dopo il nome dell'*array* come in voti[1], voti[2] ecc.

Ecco come si dichiara in Pascal una variabile *array* adatta a contenere 32 interi:

```
var                                (* NOTA. sarebbe meglio mettere una costante invece del 32:
 voti: array[1 .. 32] of integer;            ... array[1 .. NUM_ALU] of integer *)
```

Per poter passare gli *array* ai sottoprogrammi è però meglio abituarsi fin da subito a dichiarare prima un nuovo tipo di dati e solo dopo le variabili di quel tipo:

NOTA:** la sezione **type** deve essere scritta prima della sezione **var

```
type
  vet_int = array[1 .. 32] of integer;  (* vet_int sta per 'vettore di interi' *)

var
 voti: vet_int;
```

OSSERVAZIONI

- Quando si dichiara un tipo non si usano i due punti ma l'uguale (… = array …)
- Nella sezione *var* è poi possibile dichiarare tutte le variabili di quel tipo che si vogliono: tutte corrisponderanno a contenitori per 32 interi

Voti è di tipo *vet_int*, cioè un *array[1 .. 32] of integer*, esattamente come prima. Anche un parametro di un sottoprogramma potrà essere dichiarato di tipo *vet_int* (la dicitura *array[1 ..32] of integer* non sarebbe infatti accettata come tipo di un parametro).

Una volta specificata la posizione tra parentesi quadre la scrittura indica esattamente una variabile del tipo contenuto nel vettore. Detto con altre parole: *vet[7]* è, a tutti gli effetti, una variabile *integer* e può essere usata dove siete abituati ad usare una qualsiasi altra variabile *integer*:

```
Writeln('Dimmi un valore e lo memorizzerò nella posizione 4 del vettore');
readln( voti[4] );
```

```
writeln('Ora visualizzo il contenuto del sesto elemento del vettore: ');
writeln( voti[6] );
```

```
if vet[2]>x then ...
```

```
repeat
  ...
until x>vet[5];
```

```
x:=6 * vet[3] – 2*vet[7]
```

```
writeln ('Dimmi una posizione e visualizzerò il contenuto di quell'elemento: ');
readln (posizione);
writeln ( voti[posizione] );
```

L'ultimo esempio è particolarmente interessante: invece di specificare un numero preciso, come indice è possibile mettere una variabile *integer.* Questo dà al programmatore la possibilità di scandire, passare in rassegna, più elementi del vettore cambiando in un ciclo il valore della variabile indice:

```
for i:=1 to num_voti do
  writeln( voti[i] );
```

Cercate di capire bene questo spezzone di codice: è alla base di tutte le elaborazioni con gli *array* che vi saranno spiegate in questo corso!

La variabile *i*, grazie al ciclo, assume tutte le posizioni del vettore e di volta in volta individua nel ciclo elementi successivi, uno dopo l'altro dal primo all'ultimo. La prima volta che viene eseguito il ciclo la *i* vale 1 e quindi la *writeln* corrisponde a *writeln(voti[1])* e viene quindi visualizzato il contenuto del primo elemento del vettore. Poi la *i* diventa 2, e viene elaborato il secondo elemento e così via fino all'ultimo indicato nel ciclo (num_voti).

E' arrivato il momento di mettere tutto insieme e risolvere elegantemente il problema di partenza ...

```
Program medie;
const
  MAX_ALUNNI=32;
type
  vettore_32_interi = array[1 .. MAX_ALUNNI] of integer;
var
  voti: vettore_32_interi;
  i, somma_voti, num_alunni, quanti_sopra_media: integer;
  media: real;

begin
  somma_voti:=0; media:=0; quanti_sopra_media:=0;

  writeln('Quanti alunni ci sono? ');
  readln(num_alu);

 (* CARICAMENTO DEI DATI NEL VETTORE *)
  for i:=1 to num_alu do
  begin
    writeln('Inserisci voto prossimo alunno: ');
    readln(voti[i]);
    somma_voti:=somma_voti +  voti[i]
  end;

  media:=somma_voti / num_alu;

 (* I DATI SONO ANCORA NEL VETTORE: CONTROLLIAMO TUTTI *)
 for i:=1 to num_alu do
 if voti[i]>media then
    quanti_sopra_media:= quanti_sopra_media + 1;

 writeln('Risultato: ',quanti_sopra_media);
end.
```

OSSERVAZIONI

a) Se gli alunni diventassero anche 10000 il codice rimarrebbe identico! Sarebbe sufficiente modificare il valore della costante *MAX_ALUNNI*

b) Non è obbligatorio usare tutte le posizioni del vettore: al massimo ce ne sono disponibili *MAX_ALUNNI* ma il valore comunicato al computer (*num_alu*) potrebbe anche essere inferiore. A dire la verità, il programma dovrebbe controllare che questo valore sia compreso tra 1 e *MAX_ALUNNI* perché

tentare di usare un elemento prima dell'inizio del vettore o dopo la fine è un gravissimo errore che spesso porta al blocco del programma ed eventualmente del computer stesso

Caso di studio 2: MENU

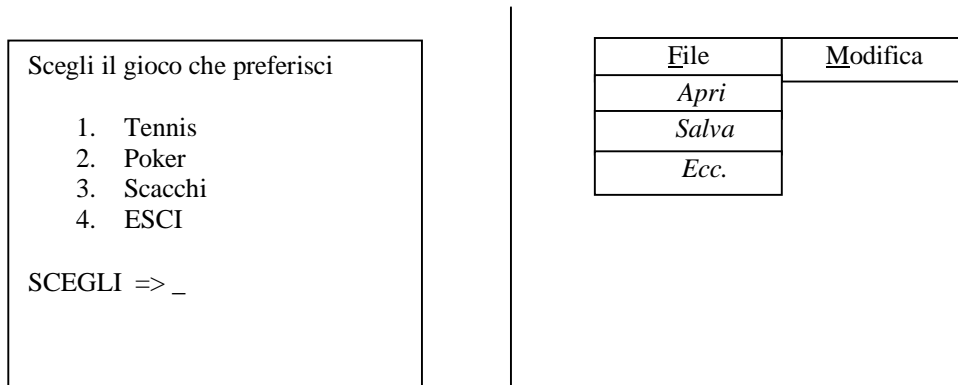
Attraverso questo esempio imparerete a passare i vettori ai sottoprogrammi.

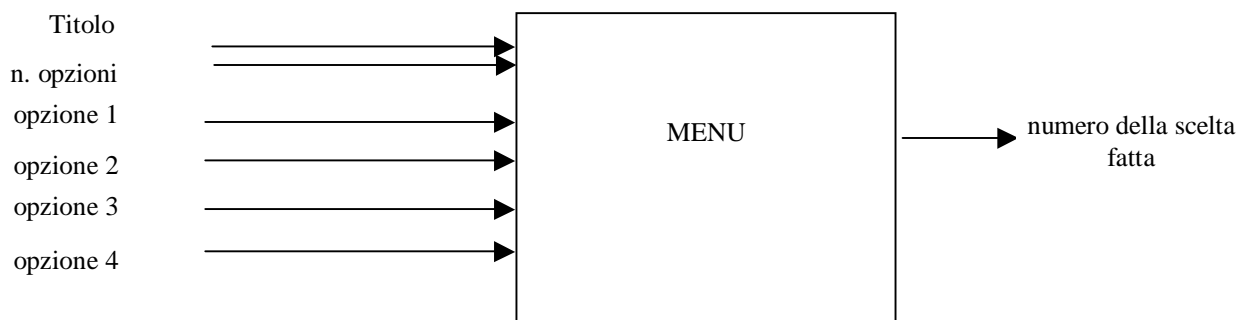
Un menu presenta sullo schermo un elenco di opzioni delle quali una sola sarà quella scelta dall'utente. Un menu può presentarsi a tutto schermo, come nell'esempio qui sotto a sinistra, o sotto forma di barra dei menu, come nell'esempio qui sotto a destra.

Scegli il gioco che preferisci

1. Tennis
2. Poker
3. Scacchi
4. ESCI

SCEGLI => _

File	Modifica
Apri	
Salva	
Ecc.	

Chiunque abbia provato ad usare un programma per computer sa che sono veramente tante le situazioni in cui è necessario dialogare con l'utente presentando un menu di scelte.

Limitandoci alla tipologia rappresentata qui sopra a sinistra, fondamentalmente lo scopo di un menu è sempre lo stesso: scrivere un titolo, sotto di questo scrivere un certo numero di scelte e leggere la risposta dell'utente. Situazione perfetta per la scrittura di un sottoprogramma: comunico titolo, opzioni, numero di queste ultime e come valore di ritorno ci si aspetterà il numero della scelta fatta. Naturalmente il sottoprogramma provvederà anche a rifiutare scelte impossibili (valori minori di 1 o più grandi del numero delle opzioni previste), a visualizzare messaggi di errore e a far ripetere la scelta fino a che l'utente non indicherà forzatamente una delle opzioni previste:

La chiamata alla funzione *menu* che corrisponde all'esempio cui ci stiamo riferendo sarebbe allora:

scelta:=menu('Scegli il gioco che preferisci',4,'Tennis','Poker','Scacchi','ESCI')

Un'altra situazione tipica è quella in cui, segnalando un errore, si vuole chiedere all'utente come procedere. Immaginiamo che si sia verificato un errore di registrazione di un documento:

scelta:=menu('Errore di scrittura su disco',3,'Riprova','Ignora','annulla')

Scusi prof., mi sa che ha dimenticato un parametro nel secondo esempio (lo studente medio finge clamorosamente di essere disinteressato a quello che scrivi, ma prova a sbagliare una virgola …): devono essere sei (il titolo, numero opzioni e le quattro scelte).

E già, ma se le scelte sono solo 3 come nel secondo esempio? O se ne servissero 12?? L'unica soluzione ora alla vostra portata sarebbe quella di prevedere il massimo numero possibile di scelte, ignorando quelle che di volta in volta non servono; la dichiarazione dell'intestazione di questa funzione sarebbe un vero 'mostro' che potrebbe assomigliare alla seguente:

```
Function menu(titolo: string; num_opzioni: integer;
                    scelta1,scelta2,scelta3,scelta4,…,sceltaN: string): integer;
```

dove N rappresenta il massimo numero di scelte utilizzabile; se le opzioni fossero di più la situazione non sarebbe gestibile a meno di modificare il sottoprogramma.

Chiamare questa funzione sarebbe veramente una 'pena'. Immaginiamo infatti di voler chiedere una conferma all'utente (in questo caso le opzioni sarebbero solo due: sì o no):

scelta:=menu('Sei sicuro',2,'Si','No','','','','','','',''…'')

anche se le opzioni sono solo due, TUTTI i parametri devono comunque essere sempre indicati (qui si immagina che quelli inutili siano indicati come stringa nulla (''))

Anche il codice della funzione sarebbe piuttosto pesante ed intricato:

```
Function menu(titolo: string; num_opzioni: integer;
                    scelta1,scelta2,scelta3,scelta4,…,sceltaN: string): integer;
begin
  writeln(titolo);  (* e fino a qui …*)
```

e qui immaginatevi, un po' come nel caso della media, tutta una serie di if … then che a seconda del numero di opzioni specificato vanno o non vanno a stampare la variabile stringa che corrisponde a quell'opzione; per semplicità mi limito solo ad iniziare …

```
writeln('1 – ',scelta1);
if num_opzioni>1 then
  writeln('2 – ',scelta2);
if num_opzioni>2 then
  writeln('3 – ',scelta3);
ecc.
```

ED ECCO LA SOLUZIONE CON GLI ARRAY …

La soluzione che sfrutta gli *array* prevede invece di passare le opzioni sotto forma di vettore di stringhe:

```
type vettore_stringhe = array[1 .. 20] of string;

Function menu(titolo: string; num_opzioni: integer; scelte: vettore_stringhe): integer;
var i,scelta: integer;
begin
writeln(titolo);
for I:=1 to num_opzioni do
 writeln(I, ' – ', scelte[i]);
```

Stampa tutte le opzioni precedute dalla I e da un trattino: 1 – opzione1, 2 – opzione 2 ecc.

```
    writeln('Scegli -> ');
    readln(scelta);
    menu:=scelta
   end;
```

Per semplicità non è stato controllato l'input dell'utente (che potrebbe essere <1 o >num_opzioni)

OPERAZIONI TIPICHE CON GLI ARRAY

In tutti gli esempi che seguono si immaginano valere le seguenti dichiarazioni:

```
const
  MAX_ELEMENTI=100;
type
  vettore_interi = array[1 .. MAX_ELEMENTI] of integer;
var
  vettore: vettore_interi; N: integer; (* N è il numero di elementi da usare *)
```

1. Caricamento di un vettore di N elementi, con N < MAX_ELEMENTI

```
for i:=1 to N do
begin
  writeln('Inserire elemento n. ', i)
  readln( vettore[i] )
end;
```

NOTA: caricare un vettore di stringhe, piuttosto che di reali comporterebbe solo la modifica delle dichiarazioni nella sezione type.

Vediamo la stessa operazione fatta con un sottoprogramma. Poiché non intendiamo avvantaggiarci di un valore restituito da una funzione, useremo una **procedura**.

Essa riceverà il vettore da caricare per indirizzo: diversamente non potremmo modificare in modo permanente i valori degli elementi del vettore passato alla procedura. Il secondo parametro corrisponde al numero di elementi da caricare (chi usa la procedura deve assicurarsi di non passare come numero di elementi un valore superiore alla capacità massima del vettore).

```
procedure carica_vettore(VAR vet: vettore_interi; N: integer);
var i: integer;
begin
  for i:=1 to N do
  begin
    writeln('Inserire elemento n. ', i)
    readln( vet[i] )
end;
end;
```

NOTA
Qualcuno potrebbe obiettare che la procedura risulta costituita da un numero di righe di codice superiore rispetto alla soluzione che non usa i sottoprogrammi. Ricordiamoci però che la procedura potrà essere chiamata tutte le volte che si vuole per caricare vettori dello stesso tipo (ANCHE in programmi diversi: basta copiarla/incollarla od includerla in una libreria). Qualche volta si tratterà di voti, altre volte di temperature, pesi od altezze ecc. La procedura non conosce il significato degli interi che sta caricando: svolge in modo anonimo il suo compito: riempire di interi un vettore che riceve come parametro, punto.

Suggerimento: prova a rendere ancora più generale la procedura prevedendo un terzo parametro corrispondente ad un messaggio personalizzato che solleciti l'utente in modo appropriato. In questo modo invece del generico 'Inserire elemento n.', potranno essere usati messaggi tipo 'Inserire la temperatura n.', o 'Inserire il peso dell'oggetto n.' ecc.

2. Visualizzazione degli elementi di un vettore di N elementi, con N < MAX_ELEMENTI

```
for i:=1 to N do
  writeln( vettore[i] );
```

Procedura per visualizzare gli elementi di un vettore. Riceve il vettore (per valore, visto che non dobbiamo modificare in alcun modo il suo contenuto!) ed il numero di elementi da considerare.

NOTA: chi usa la procedura deve assicurarsi di non passare come numero di elementi un valore superiore alla capacità massima del vettore.

```
procedure visualizza_vettore(vet: vettore_interi; N: integer);
var i: integer;
begin
  for i:=1 to N do
      writeln( vet[i] )
end;
```

3. Somma dei valori degli elementi di un vettore di N elementi, con N < MAX_ELEMENTI

```
somma:=0;

for i:=1 to N do
  somma:=somma + vettore[i] ;
```

Funzione per la stessa operazione.

```
function  somma_vettore(vet: vettore_interi; N: integer):integer;
var tot: integer;
begin
  tot:=0;
  for i:=1 to N do
      tot:=tot + vettore[i] ;
 somma_vettore:=tot
 end;
```

4. Media degli elementi di un vettore di N elementi, con N < MAX_ELEMENTI

NOTA: questo esempio è particolarmente interessante se utilizziamo i sottoprogrammi. Infatti la funzione che calcola la media può richiamare quella, già scritta, che calcola la somma!

```
somma:=0;

for i:=1 to N do
  somma:=somma + vettore[i] ;

media:= somma / N;
```

Funzione per la stessa operazione.

```
function  media_vettore(vet: vettore_interi; N: integer):integer;
begin
   media_vettore:= somma_vettore(vet,N) div N
end;
```

NOTA: un sottoprogramma può passare ad un altro i suoi stessi parametri, con le stesse modalità viste sino ad ora (per valore o per indirizzo).

5. Posizione dell'elemento con valore minimo in un vettore di N elementi, con N < MAX_ELEMENTI

L'algoritmo è basato su questo ragionamento: all'inizio ipotizzo che il minimo sia l'elemento che si trova in posizione 1. Quindi controllo tutti gli altri elementi, dal secondo in poi, e tutte le volte che trovo una posizione con un valore più piccolo diventa quest'ultima la nuova posizione del minimo.

```
posizione_min:= 1;

for i:=2 to N do
 if  vettore[i] < vettore[posizione_min] then   (* trovata posizione i con valore più piccolo *)
   posizione_min := i;
```

Funzione per la stessa operazione.

```
function posizione_min(vet: vettore_interi; N: integer):integer;
var pos_min:integer;
begin
  pos_min:= 1;

  for i:=2 to N do
  if  vet[i] < vet[pos_min] then   (* trovata posizione i con valore più piccolo *)
   pos_min := i;

  posizione_min := pos_min
end;
```

NOTA: se nel vettore ci sono più elementi minimi con lo stesso valore, viene individuato il primo (quello con indice minore). Infatti quando viene confrontato il primo minimo con gli altri la condizione *vet[i] < vet[pos_min]* non è verificata e l'aggiornamento della posizione non viene effettuata. Aggiungendo l'uguaglianza al confronto (*vet[i] <= vet[pos_min]*) verrebbe invece localizzato l'ultimo minimo (quello con indice superiore).

6. Posizione dell'elemento con valore massimo in un vettore di N elementi, con N < MAX_ELEMENTI

E' sufficiente cambiare al punto precedente *posizione_min* con *posizione_max*, < con > e *pos_min* con *pos_max* per ottenere l'algoritmo desiderato.

7. Ricerca della posizione di un valore in un vettore di N elementi, con N < MAX_ELEMENTI

```
writeln('Quale elemento cerchi? ');
readln(cercato);

posizione:=0;

for i:=1 to N do
  if vettore[i] = cercato then
     posizione:=i;
```

OSSERVAZIONI

- Se l'elemento non viene trovato la variabile posizione rimarrà 0. E' importante controllarla prima di usarla! Ad esempio: *if posizione>0 then …(* elemento trovato, possiamo procedere *)*
- Se nel vettore l'elemento cercato appare più volte, il ciclo *for* localizza l'ultimo (comunque la scansione arriva ad N); per localizzare il primo è necessario controllare 'manualmente' la ricerca usando un ciclo *repeat* (come mostrato più avanti)

```
function cerca_vettore(vet: vettore_interi; N: integer; cercato: integer):integer;
var posizione: integer;
begin
  posizione:=0;

  for i:=1 to N do
    if vet[i] = cercato then
       posizione:=i;

cerca_vettore:=posizione
end;
```

Ed ecco la soluzione con il *repeat*, nel caso si voglia il primo elemento uguale a quello che si sta cercando. Questa soluzione può essere anche molto più efficiente di quella con il *for*. Ad esempio se gli elementi del vettore fossero diecimila e quello che si sta cercando fosse nella quarta posizione, il ciclo *for* andrebbe inutilmente a controllare gli altri 9996, mentre il ciclo *repeat* si fermerebbe al quarto!

```
posizione:=0; i:=0;

repeat
  i:=i+1;
  if vettore[i] = cercato then
     posizione:=i
until (i=N) or (posizione<>0);
```

```
function cerca_vettore(vet: vettore_interi; N: integer; cercato: integer):integer;
var posizione,i: integer;
begin
  posizione:=0; i:=0;

  repeat
    i:=i+1;
    if vettore[i] = cercato then
       posizione:=i
  until (i=N) or (posizione<>0);

cerca_vettore:=posizione
end;
```

7. Ordinamento dei valori in un vettore di N elementi, con N < MAX_ELEMENTI
   METODO PER SCAMBIO – BUBBLE SORT

Quello che sto per presentare è un metodo non molto efficiente quando il numero degli elementi supera il centinaio, ma ha il pregio di essere breve, relativamente semplice ed abbastanza facile da ricordare. Appartiene alla famiglia degli algoritmi di ordinamento (SORT) detti per scambio perché adotta una scansione sistematica che confronta 'ciecamente' tutti gli elementi e scambia quelli fuori posto tra loro. Lo scambio di elementi è usato in modo pesante: altri algoritmi adottano tecniche che 'ragionano' di più sulla situazione del vettore od adottano tecniche più intelligenti (ripagate solo se il numero degli elementi è sufficientemente elevato) diminuendo il numero di scambi necessari. Ovviamente questi algoritmi sono più difficili …

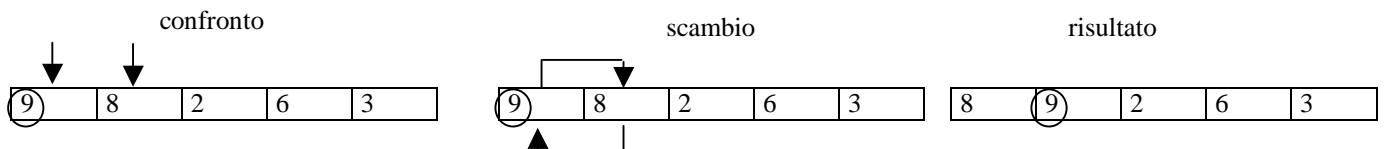
Prima di esaminare il codice Pascal conviene presentare l'algoritmo in modo 'informale' aiutandosi anche con una simulazione grafica. Come punto di partenza consideriamo il seguente vettore disordinato:

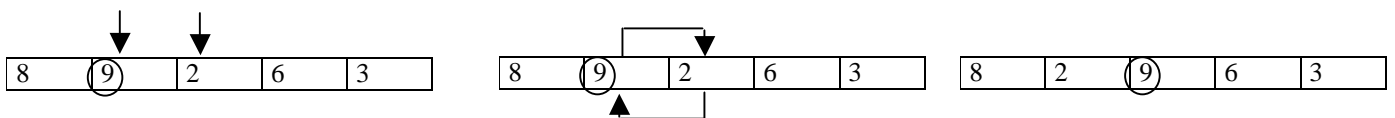
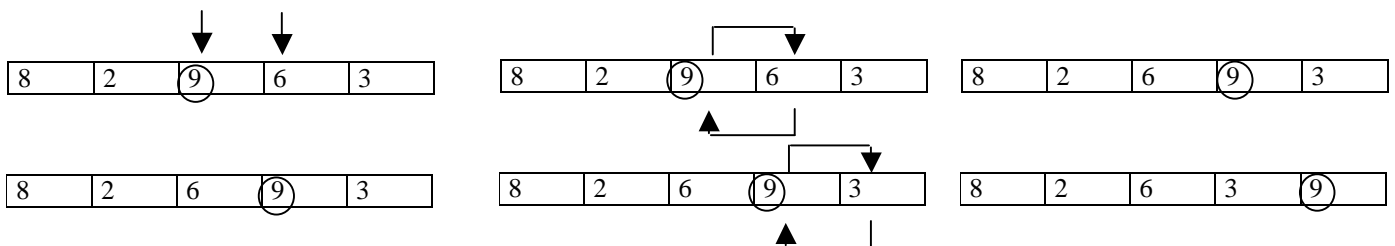
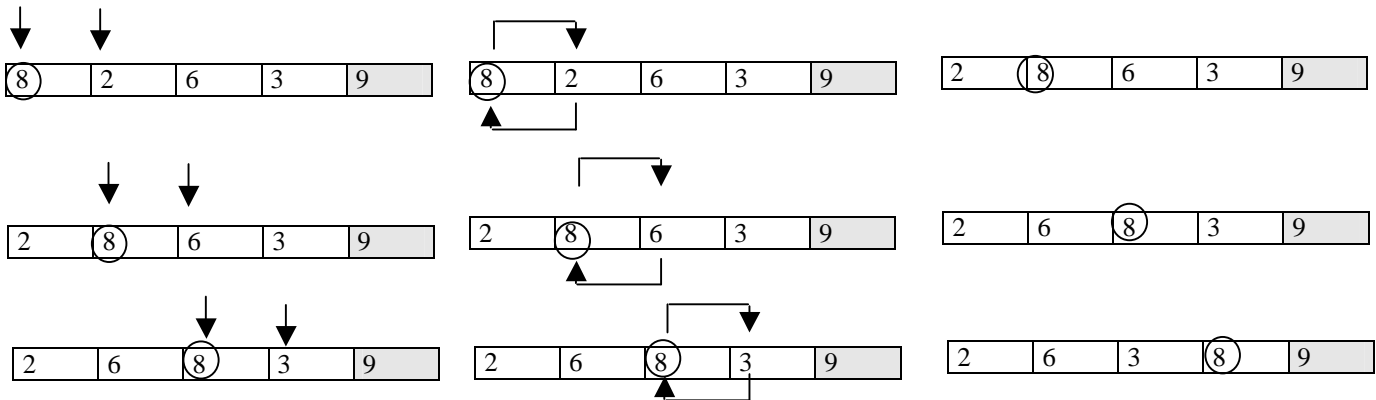
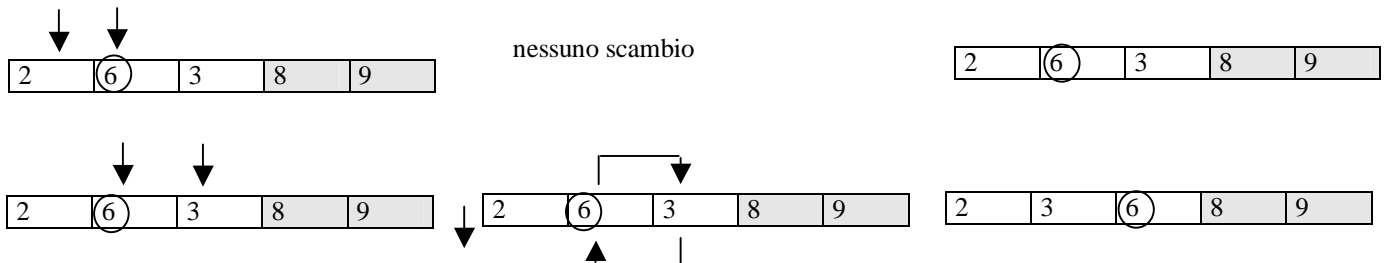
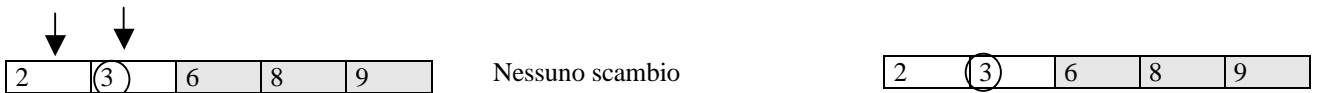
9	8	2	6	3

— ordinamento →

2	3	6	8	9

L'algoritmo procede in questo modo:

1. confronta la prima posizione con la seconda: se il numero contenuto nella prima posizione è maggiore di quello nella seconda scambia i numeri:

NOTA: tenete d'occhio il numero 9; come una bolla d'aria nell'acqua risalirà verso la 'superficie' (la parte finale del vettore), da cui il nome dell'algoritmo (bubble=bolla sort=ordinamento)

2. spostati di una posizione verso destra e ripeti il confronto/scambio

3. ripeti il punto 2 fino a raggiungere la penultima posizione (che verrà confrontata con l'ultima)

Dopo questo primo 'giro' (passata) l'elemento più grande del vettore (il 9) ha raggiunto la sua giusta posizione. Ora si ripete il tutto fermandoci però alla penultima posizione (l'ultima, come detto prima, è già ordinata e va lasciata stare!). Per ricordarcelo, sfumiamo in grigio l'ultima casella del vettore.

Ora ripetiamo ma fermandoci alla terzultima posizione (le ultime due sono in ordine)

NOTA: il vettore sarebbe già in ordine, ma l'algoritmo 'non se ne accorge' e, ciecamente, continua i confronti:

2	3	6	8	9

l'algoritmo si ferma quando ha ripetuto il processo n-1 volte

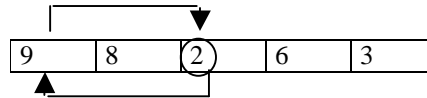
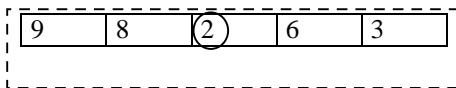
Ed ecco la codifica in pascal:

```
(* per n – 1 volte ... *)
for i:=1 to n - 1 do

(* parti dalla prima posizione e confronta
   tutte le coppie fino alla n – i *)
  for j := 1 to n - i do

(* se trovi due celle adiacenti non
  in ordine, scambiale *)

     if vettore[j] > vettore[j+1] then
     begin
        tmp:=vettore[j];
        vettore[j]:=vettore[j+1];
        vettore[j+1]:=tmp
     end;
```

```
procedure bubble_sort(VAR vet: vettore_interi; n: integer);
var i,j: integer;
begin
 for i:=1 to n - 1 do
  for j := 1 to n - i do
    if vet[j] > vet[j+1] then
   begin
      tmp:=vet[j];
      vet[j]:=vet[j+1];
      vet[j+1]:=tmp
    end
end;
```

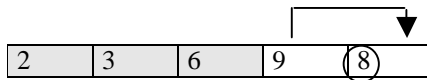
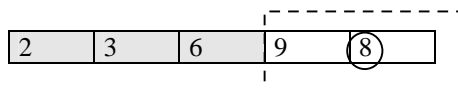
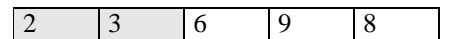
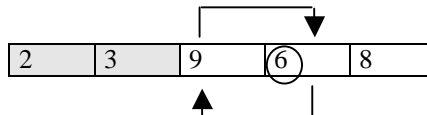
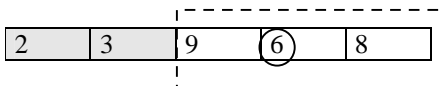
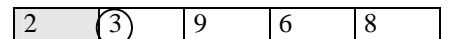
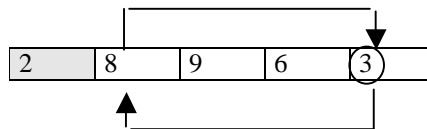
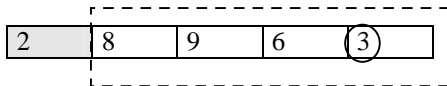
7. Ordinamento dei valori in un vettore di N elementi, con N < MAX_ELEMENTI
METODO PER SELEZIONE – PER RICERCA DEL MINIMO

Questo algoritmo appartiene alla categoria chiamata ‘per selezione’ perché invece di scambiare in continuazione celle adiacenti tra loro, cerca il miglior numero da spostare, riducendo di molto il numero degli scambi. In dettaglio:

1. Partendo dalla prima posizione, localizza la cella con valore minimo; prendi il minimo e mettilo in prima posizione; il numero in prima posizione mettilo nella cella in cui hai trovato il minimo:

2. Spostati in avanti di una casella ed ignora la prima (già in ordine): cerca il minimo nella parte del vettore ‘avanzata’; mettilo in seconda posizione; il numero in seconda posizione mettilo nella cella in cui hai trovato il minimo. Ripeti il procedimento fino alla penultima posizione (a quel punto il numero in ultima posizione è per forza in ordine.

2 8 9 6 3   2 8 9 6 3   2 3 9 6 8

2 3 9 6 8   2 3 9 6 8   2 3 6 9 8

2 3 6 9 8   2 3 6 9 8   2 3 6 8 9

Ed ora il codice pascal:

```
for i:=1 to n-1 do
begin
  pos_min:=i;

(* cerco la posizione del minimo
    tra i e ultimo elemento *)
  for j:=i+1 to n do
     if vet[j] < vet[pos_min] then
        pos_min:=j;

   (* scambio con il minimo *)
   tmp := vet[i] ;
   vet[i] := vet[pos_min];
  vet[pos_min]:=tmp
end;
```

```
procedure selezione_sort(VAR vet: vettore_interi; n: integer);
var i,j: integer;
begin
  for i:=1 to n-1 do
  begin
    pos_min:=i;

    for j:=i+1 to n do
      if vet[j] < vet[pos_min] then
          pos_min:=j;

    tmp := vet[i] ;
    vet[i] := vet[pos_min];
    vet[pos_min]:=tmp
  end
end;
```

INTRODUZIONE ALL'USO DEI RECORD

I dati di un vettore o di una matrice devono essere tutti uguali: o tutti integer, o tutti real o string ecc. Questo limite porta a notevoli complicazioni anche per situazione semplici. Immaginiamo infatti di voler rappresentare la seguente scheda:

```
Persona
  Cognome
  Nome
  Eta
Fine-Persona
```

Rossi	Verdi	Bianchi

Mario	Giovanni	Sandra

21	34	15

Soluzione con tre vettori

Rossi	Verdi	Bianchi
Mario	Giovanni	Sandra

21	34	15

Soluzione con una matrice ed un vettore

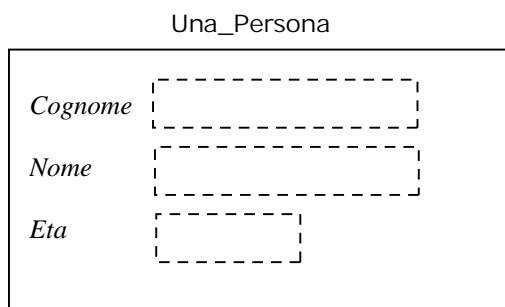
Usando gli array sono possibili due soluzioni:

1. Tre vettori in parallelo: uno per i cognomi, uno per i nomi ed uno per le età;

2. Una matrice (di stringhe) a due colonne (la prima per il cognome e la seconda per il nome) ed un vettore per le età.

Gestire vettori o matrici e vettori in parallelo non è molto agevole ... Per fortuna molti linguaggi di programmazione consentono al programmatore di definire nuovi tipi di dato, strutturati come meglio si crede, e di dichiarare poi variabili di quel tipo. Uno di questi tipi di dato è il record.

Vediamo l'esempio della scheda anagrafica visto poc'anzi:

```
program ProvaRecord;

Type
 TPersona = record
   Cognome: string;
   Nome: string;
   Eta: integer
 End;
...
Var
 Una_Persona: TPersona;
```

Il nuovo tipo si chiama TPersona (la T, per nulla obbligatoria, aiuta comunque a ricordare che si tratta di un 'T'ipo). I nuovi tipi devono essere definiti in una sezione specifica, dopo le costanti: la sezione type. Potete naturalmente definire anche più di un tipo. Il tipo TPersona contiene tre valori (chiamati campi del record): cognome, nome ed eta (con i nomi delle variabili è sempre meglio evitare gli accenti...).

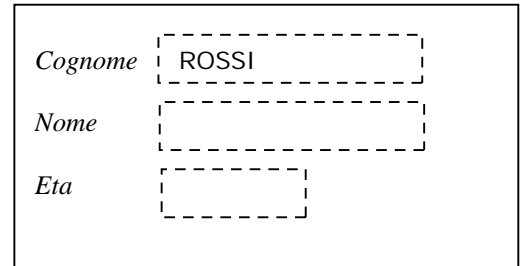
Non è possibile lavorare direttamente con i tipi (non potete ad esempio scrivere integer:=3 !) ma è necessario dichiarare variabili di quel tipo. Nella sezione var trovate quindi la variabile Una_Persona di tipo TPersona.

Qui sopra potete vedere, a fianco della definizione del nuovo tipo, una rappresentazione grafica del record: l'intero rettangolo esterno è il record; all'interno i componenti. Ma come si 'riempiono' di dati le caselle tratteggiate?

Ecco il comando per memorizzare la stringa 'Rossi' nel campo cognome:

```
Una_Persona . cognome := 'ROSSI';
```

QUINDI: si inizia con il nome della variabile record (Una_Persona), si mette un punto, e si indica il campo da usare (cognome).

Similmente si potrebbe procedere con il nome e l'età.

I campi di un record sono a tutti gli effetti variabili come quelle che già conoscete. E' quindi possibile scrivere:

```
readln(Una_Persona.nome);        (* lettura da tastiera *)
  ...
writeln('La persona si chiama: ', Una_Persona.Cognome, ' – ', Una_Persona.nome);
  ...
if Una_Persona.cognome='Rossi' then ...
```

NOTA: dimenticarsi il nome della variabile record (a meno di utilizzo del with, spiegato più avanti) è un grave errore:

cognome:='rossi'   DI QUALE PERSONA ??? ci potrebbero essere tante variabili di tipo TPersona ...

Il costrutto with

Immaginiamo di avere una variabile record con il nome piuttosto lungo e con molti campi all'interno. E' scomodo riscrivere il nome della variabile tutte le volte che si vuole usare un suo campo; se poi questi ultimi sono molti è ancora più scomodo. Il Pascal mette a disposizione un costrutto (with, cioè con ...) che consente di scrivere una volta sola il nome della variabile record finchè non si ha finito di usare i suoi campi:

```
with Una_Persona do
begin
  readln(cognome);
  readln(nome);
  readln(eta)
end;
```

Tra il *begin* e *l'end* del costrutto *with* invece di scrivere:

Readln(**Una_Persona.**cognome**);**

è sufficiente scrivere:

Readln(**cognome**);

Vettori di record

Una famiglia è fatta di più persone: Una_Famiglia: array[1..3] of TPersona;

Ecco come dare un valore al primo record del vettore:

```
Una_Famiglia[1].cognome:='Giorgio';
Una_Famiglia[1].eta:='Albenghi';
Una_Famiglia[1].eta:=43;
```

Attenzione! Il seguente è un errore piuttosto comune:

Una_Famiglia.eta[3]:=23

Sarebbe il field *eta* ad essere considerato un vettore!

Quindi una_famiglia è prima di tutto un vettore: dopo il suo nome è necessario indicare quale elemento si intende usare (una_famiglia[1]). Poi, essendo ogni elemento del vettore un record si continua con la sintassi di questi ultimi (un punto ed il nome del campo da usare).

Ecco un esempio interessante sul come usare il costrutto with con i vettori di record. Si tratta di un ciclo di caricamento del vettore della famiglia:

```
For i:=1 to num_componenti do
   With Una_Famiglia[i] do            (* con l'i-mo componente fai ... *)
   begin
    Readln(cognome);
    Readln(nome);
    Readln(eta)
  end
```

Record di record

Un field di un record può tranquillamente essere a sua volta un record: definiamo prima un record per le date e poi un record per i compleanni (che conterranno il record della data):

```
Type
  TData=record  (* un record normale *)
    Gg,mm,aa: integer;
  End;

  TCompleanno=record
    Data: TData;                 (* il record Data è contenuto nel record TCompleanno *)
    Nominativo: string;
  End;

Var
  MioCompleanno: TCompleanno;
```

accesso ai componenti senza with

```
MioCompleanno.Data.gg := 30;
MioCompleanno.Data.mm := 10;
MioCompleanno.Data.aa := 1963;
MioCompleanno.Nominativo:= 'Fabrizio Camuso';
```

accesso ai componenti con with

```
with MioCompleanno do
begin

  with Data do
  begin
    gg:=30; mm:=10; aa:=1963
  end

  Nominativo:='Fabrizio Camuso'
end;
```

Attenzione. Sarebbe un errore:

```
  with MioCompleanno do
  begin
    gg:=30; mm:=10; aa:=1963;
  end
```

perchè gg, mm ed aa non sono direttamente campi di MioCompleanno ma del record Data in esso contenuto

ESEMPIO : scrivere un programma per la gestione di una biblioteca che preveda la ricerca dei dati di un libro per autore, o per titolo o per editore

```
program libri;
uses crt;

const
  MAX_LIBRI=50;

type
     TLibro=record
      codice: integer;
      autore,titolo,editore: string;
    end;

var
  biblioteca: array [1..MAX_LIBRI] of TLibro;
  numero_libri,cont,scelta: integer;  da_cercare: string;

begin
  numero_libri:=0;cont:=0;scelta:=0;da_cercare:='';
  clrscr;

repeat
    write('Quanti sono i libri (max. ',MAX_LIBRI,') ? ');
    readln(numero_libri);

    if (numero_libri<1) or (numero_libri>MAX_LIBRI) then
      writeln('Valore non accettabile, riprova ...')

  until (numero_libri>0) and (numero_libri<=MAX_LIBRI);

  (* carico i dati nel vettore di record *)
  for cont:=1 to numero_libri do
      with biblioteca[cont] do
      begin
        write('Inserire il titolo del libro n. ',cont,' : ');
        readln(titolo);
        write('Inserire l''autore del libro ',titolo,' : ' );
        readln(autore);
        write('Inserire l''editore del libro ',titolo,' : ' );
        readln(editore);
        write('Inserire il codice del libro ',titolo,' : ' );
        readln(codice);
      end;
```

```
 repeat
   clrscr;
   writeln('RICERCA CODICI LIBRI PER ...');
   writeln('1 - Ricerca per autore');
   writeln('2 - Ricerca per titolo');
   writeln('3 - Ricerca per editore');
   writeln('4 - Fine operazioni');
   writeln;

   repeat
     write('Scegli un opzione: ');
     readln(scelta);

     if (scelta<1) or (scelta>4) then
       writeln('Scelta non corretta');

    until (scelta>0) and (scelta<5);

    if scelta<>4 then
    begin
       case scelta of
        1: write('Inserire l''autore interessato: ');
        2: write('Inserire il titolo interessato: ');
        3: write('Inserire l''editore interessato: ');
       end;

      readln(da_cercare);

      for cont:=1 to numero_libri do
        with biblioteca[cont] do
        begin
         case scelta of
           1: if da_cercare=autore then writeln(codice);
           2: if da_cercare=titolo then writeln(codice);
           3: if da_cercare=editore then writeln(codice);
         end;

        writeln('Premere INVIO per continuare ...');
        readln
      end
    until scelta=4
end.
```

I FILE – GESTIONE TRADIZIONALE (non DBMS)

Il file (archivio) è una collezione di dati registrati su un supporto di massa (floppy disk, hard disk, CD, DVD, nastro ecc.). Da un punto di vista fisico tutti i file sono memorizzati come una sequenza di byte: è il modo con cui questi ultimi sono interpretati che fa la differenza. Ad esempio, i 1000 byte che costituiscono un certo file potrebbero essere intesi come dieci schede (record) da 100 byte con le informazioni di un cliente della ditta. E di ciascun record i primi 10 byte potrebbero rappresentare il codice del cliente, i successivi 30 il nominativo ecc. (i cosiddetti 'fields', campi). Ma, in un contesto completamente diverso 1000 byte potrebbero rappresentare la codifica binaria di un'immagine o di un suono.

Sui supporti, i byte sono di solito trattati a blocchi (cioè non viene mai letto/scritto un solo byte alla volta) secondo una tecnica di suddivisione dello spazio a disposizione che dipende dal supporto (su un floppy disk tracce concentriche suddivise in 'spicchi' ad individuare settori, su un CD un'unica traccia a spirale, sul nastro bande magnetiche perpendicolari o elicoidali ecc.). Le letture/scritture avvengono in aree di transito in RAM chiamate buffer(s). Quando si leggono dati da un dispositivo, uno o più blocchi vengono memorizzati nel buffer di lettura e da lì prelevati man mano che l'applicazione ne fa richiesta (in modo da non effettuare inutili letture fisiche dei supporti, in quanto probabilmente a più richieste di lettura di dati da parte dell'applicazione corrisponderà una sola lettura di dati dal disco al buffer). Similmente i dati da registrare su file sono prima accumulati in un buffer di scrittura fino al riempimento di quest'ultimo e solo al raggiungimento di questa situazione vengono fisicamente scritti sul disco (evitando in questo modo che ad ogni richiesta di scrittura da parte dell'applicazione avvenga una dispendiosa scrittura fisica sul disco). Ogni dispositivo ha poi particolarità legate alla sua natura: un hard disk può essere formato da più piatti ed il relativo drive avere molte testine di lettura/scrittura, un floppy ha invece un solo piatto (un film flessibile e sottile come un capello a forma di cerchio) ed una sola coppia di testine.

Riassumendo, la logica di controllo di questi dispositivi è assai diversificata, sia perché l'accoppiata supporto e relativo drive è formata da dispositivi elettromeccanici diversi, sia perché diverse sono le tecniche con cui si può decidere di usarli per disporre dello spazio di memorizzazione (quante tracce gestire sul floppy? come organizzare la spirale di bit su di un CD?).

Agli albori dell'informatica la gestione dei supporti era sotto la completa responsabilità dei programmatori, costretti a confrontarsi direttamente con l'hardware (aspetto fisico) oltre che a decidere come usare lo spazio messo a disposizione (aspetto logico). Detto in altre parole doveva occuparsi sia della gestione fisica delle informazioni che di quella logica: cioè pilotare i dispositivi ed allo stesso tempo gestire i byte in modo da poterli interpretare come informazioni utili (quali archivi creare? Come strutturarli in campi informativi? Come mettere in relazione i record un archivio con quelli di un altro?). Questa doppia 'responsabilità' porta con se parecchi problemi tra cui, mettendo in evidenza i più eclatanti, è opportuno citare i seguenti:

- Necessità di conoscere in modo approfondito l'hardware da controllare.
- Come conseguenza del punto precedente aumenta la complessità dell'attività di programmazione.
- I programmi sono molto legati all'hardware: quando quest'ultimo cambia essi vanno modificati.
- L'efficienza nella gestione dei supporti dipende dalla competenza del programmatore.
- Poca uniformità: ogni programmatore adotta tecniche diverse di allocazione/gestione dello spazio
- Notevole spreco di tempo/uomo: ogni programmatore codifica routine già codificate da altri (chi meglio, chi peggio).
- Basso livello di astrazione nella programmazione: lo sviluppatore invece di concentrarsi solo su aspetti logici legati al problema è costretto a gestire altri dettagli.

Ci sono anche i soliti vantaggi di una programmazione a basso livello:

- controllo accurato dell'hardware
- possibilità, se si è BRAVI programmatori, di ottimizzare il codice per il massimo delle prestazioni

Solo con l'avvento di sistemi operativi sufficientemente evoluti (DOS, UNIX, windows NT) il programmatore ha potuto svincolarsi dagli aspetti hardware ed in parte anche da quelli logici. Il modulo del sistema operativo chiamato 'file system', infatti:

Ø si occupa della gestione hardware (file system fisico): pilota dispositivi anche molto diversi tra loro (floppy, hard disk, CD ROM, nastri ecc.) per la registrazione e la lettura di blocchi di byte;

Ø mette a disposizione delle entità logiche chiamate file (file system logico) gestite tramite un nome ed una posizione (path, cammino) in una struttura, di solito gerarchica (directory, sotto directory ecc.); mette in oltre a disposizione comandi per creare, eliminare, spostare, rinominare i file; questi comandi possono non solo essere imparti grazie ad una riga di comando (MS DOS, Unix, Linux) o ad un ambiente a finestre (Windows, Unix, Linux) ma anche in un programma;

L'esame dettagliato del modulo file system viene svolto nel corso di Sistemi. Nel corso di Informatica interessa invece cogliere gli aspetti più legati allo sviluppo del software. Che dire allora dei linguaggi di programmazione? Come si sono evoluti in risposta a queste problematiche?

A livello di linguaggi 'assembly' il tutto si risolve attraverso chiamate a servizi del sistema operativo (nel caso di MS DOS attraverso il noto meccanismo degli interrupt software, come dovrebbe esservi chiarito nel corso di sistemi).

I linguaggi di terza generazione ('C', Pascal, Basic, Cobol ecc.) offrono istruzioni più evolute (apri un file, chiudilo, leggi dal file, scrivi su di esso, spostati ad una certa posizione ecc.) che sono poi tradotte dal compilatore negli stessi meccanismi a basso livello dell'assembly.

Da alcuni anni la gestione delle informazioni è diventata talmente importante per il successo dei sistemi informativi e le funzionalità richieste talmente sofisticate da rendere insufficienti le funzionalità offerte dai sistemi operativi e, di conseguenza, dei programmi che vi fanno affidamento. Questo fatto giustifica l'esistenza sul mercato di complessi software dedicati esclusivamente alla gestione delle basi di dati aziendali: i DBMS (data base management system: saranno oggetto di un'ampia porzione del corso di Informatica di quinta).

Vista l'esistenza dei DBMS perché parlare delle tecniche 'tradizionali' in presenza di strumenti talmente avanzati? Ecco alcuni buoni motivi:

Ø i requisiti hardware/software (microprocessore, RAM, spazio su hard disk) richiesti dai DBMS sono decisamente elevati: in più di una situazione non sono semplicemente disponibili e l'unica alternativa (spesso perfettamente adeguata) è rappresentata dalle tecniche 'tradizionali';

Ø certe elaborazioni sono molto difficili da realizzare con i linguaggi specifici per i data base: questi linguaggi 'pagano' la loro indubbia semplicità che li rende alla portata anche del non programmatore con una minore 'potenza'; il problema è così sentito che in molti linguaggi per data base sono stati reintrodotti costrutti di programmazione tipici dei linguaggi tradizionali

Ø per certe operazioni la velocità delle tecniche tradizionali è impareggiabile: nessun DBMS supera in velocità linguaggi come il Pascal o il C per scrivere e leggere file di testo (e non dimenticate che le pagine WEB in HTML sono file di testo ...)

Ø conoscere le tecniche passate aiuta a capire, apprezzare e sfruttare molto meglio ciò che le nuove mettono a disposizione.

TIPI DI FILE

1. File di testo (text file): sono sequenze di bytes con valori numerici nell'intervallo valido per la codifica dei caratteri per un certo sistema operativo. Nel caso di MS DOS / Windows (codifica ASCII) questi file sono terminati dal carattere con codice 26 (CTRL Z). La fine di un file viene indicata con la sigla EOF (End Of File, fine del file). L'inizio con BOF (Begin Of File, inizio del file)

   I file di testo sono suddivisi in righe. Ogni riga è terminata dai byte CR (Carriage Return, ritorno carrello) codice ASCII 13) e LF (Line Feed, avanzamento di riga, codice ASCII 10).

   Attenzione: le righe sono terminate o solo da CR o solo da LF o da tutti e due a seconda del sistema operativo in uso (con MS DOS/Windows le righe sono terminate da CR seguito da LF).

   Windows è in grado di trattare anche caratteri UNICODE, uno standard internazionale che consente, grazie all'uso di due byte per codificare ciascun carattere, di rappresentare set di caratteri molto ricchi. Per lo stesso motivo i linguaggi più recenti (Delphi con il suo Object Pascal ne è un esempio) sono in grado di definire variabili carattere aderenti a questo standard.

   Ecco come potremmo rappresentare graficamente un file di testo:

BOF

questa è la prima riga del file di testo. Può contener anche spazi come questi !!	CR	LF	
io sono la seconda RIGA bla bla bla bla bla!	CR	LF	
io sono la riga più corta !!!	CR	LF	
	CR	LF	
	CR	LF	
	CR	LF	
	CR	LF	EOF

EOF

   Le istruzioni per scrivere e leggere dati dai file di testo sono line-oriented nel senso che permettono di leggere o scrivere intere righe del file e non solamente una parte dei caratteri di una certa riga. Abbiamo così comandi per registrare una riga oppure per leggerla da un file. Una grossa limitazione dei file di testo in Pascal è che le righe già inserite non possono essere modificate (riscritte).

   I file testuali sono il tipo di file meno sofisticato e meno efficiente per rappresentare situazioni reali di una certa complessità ma allo stesso tempo assicurano la massima portabilità tra elaboratori con diversi sistemi operativi e tra periferiche altrimenti incompatibili. Qualsiasi linguaggio su qualsiasi piattaforma hardware/software è infatti in grado di gestire file di testo. Nel peggiore dei casi si renderà necessaria una (semplice) transcodifica (ad esempio da ASCII a EBCDIC).

   Non sono inoltre particolarmente adatti a contenere informazioni non testuali: qualsiasi codifica binaria (la rappresentazione interna di un numero in virgola mobile, il codice operativo od un operando di un'istruzione assembly, un immagine, un suono un'animazione) contenente un byte con valore 26 verrebbe considerato, inesorabilmente, EOF. Naturalmente ci sarebbe sempre il modo di codificare gli stessi dati usando solo byte diversi da EOF, CR e LF ma la soluzione risulterebbe meno efficiente e poco elegante. Meglio usare i file non testuali (vedi sotto).

2. File binari ('binary file' o non di testo): a livello fisico sono ancora formati da una sequenza di byte il cui valore non viene però interpretato come un carattere ASCII. Così, mentre un byte con valore (decimale) 65 viene interpretato come la 'A' in un file di testo, lo stesso valore potrebbe specificare una tonalità di colore di un'immagine in un file jpeg (uno dei formati grafici più diffusi).

   La fine di questi file non è segnalata dal carattere CTRL Z ma attraverso altri meccanismi (ad esempio tenendo traccia della lunghezza in byte del file). Anche i byte con valore numerico 13 e 10 (i CR e LF dei file di testo) non hanno un significato particolare.

   Le istruzioni di lettura/scrittura per questi tipi di file permettono tipicamente di leggere blocchi di byte di dimensioni specificate dal programmatore.

   Sono adatti per memorizzare qualsiasi tipo di file, testo compreso (ovviamente non potremo sfruttare le primitive di lettura e le particolarità dei file di testo). Sono file binari le traduzioni in linguaggio macchina dei programmi, immagini, suoni, archivi di basi di dati per i quali si è preferito questo formato a quello testuale ecc.

3. File tipizzati: sono file per i quali le unità informative (chiamate record in Pascal) hanno una struttura (un tipo) ben precisa. Come i record Pascal gestiti in RAM, i record su file sono comodamente suddivisi in campi (fields). Ad esempio:

```
type
  TCalciatori=Record

      matricola: integer;
      cognome: string;
      nome: string;
      ecc.
  end;

var FileCalciatori: file of TCalciatori;
```

   Le istruzioni di lettura/scrittura sono in grado di leggere/scrivere un intero record per volta, di spostarsi per leggere/scrivere in una posizione qualsiasi del file, e, operazione impossibile con i file di testo, di modificare (riscrivere) una scheda.

Nota: ribadisco che a livello fisico i file sono comunque tutti formati da una sequenza di byte; la differenziazione che ne facciamo è a livello logico. Una parte della letteratura informatica si limita a distinguere tra file di testo e non di testo (i file tipizzati vengono considerati file binari).

TIPI DI ORGANIZZAZIONE (ACCESSO)

L'organizzazione fa riferimento alle modalità con cui è possibile accedere ai dati sui supporti:

- Sequenziale: i dati possono essere letti/scritti in modo strettamente lineare; per leggere/scrivere un dato è necessario leggere i precedenti per posizionare il dispositivo di lettura/scrittura nel punto giusto; il tempo per accedere ad un dato è quindi molto dipendente dalla sua posizione sul supporto; ad esempio, per un nastro magnetico l'unico tipo di organizzazione possibile è quella sequenziale: se il nastro è all'inizio, il tempo per leggere il primo blocco di dati è assai inferiore al tempo necessario per leggere un blocco che sta a metà o alla fine del nastro.

- Random: se è possibile raggiungere un blocco qualsiasi di dati impiegando praticamente lo stesso tempo indipendentemente dalla posizione del blocco. Ad esempio il tempo necessario a pilotare una delle testine di lettura/scrittura in un punto qualsiasi della superficie di un piatto di un hard disk, partendo da una posizione qualsiasi, non è esattamente zero ma è comunque abbastanza piccolo da ritenere (quasi) identico il tempo necessario a leggere uno qualsiasi dei blocchi che costituiscono un file.

NOTA: la distinzione tra organizzazione di un file tra sequenziale e random è indipendente dalle capacità fisiche del supporto ed è stabilita dalla modalità di accesso a quel file. Ad esempio, l'hard disk a livello fisico è sicuramente random ma nulla vieta di organizzare la registrazione di un file in modo che il suo accesso sia comunque sequenziale (pensate ad un file di testo in Pascal).

Nella letteratura informatica ci si riferisce alle seguenti come ad organizzazioni a se stanti ma in realtà sono tutte costruite su quella 'random':

- Con indici: quando l'accesso ai dati è velocizzato dalla gestione di strutture (su file separati o nello stesso file dei dati) che, grazie alla loro organizzazione, consentono di risalire rapidamente alla posizione in cui trovare i dati. Il modo di realizzare l'indice può variare grandemente. Individuato un campo informativo in relazione al quale si vuole velocizzare l'accesso (ricerca per codice o per cognome, ecc.) l'indice conterrà una copia di tutti i valori contenuti nel file dati per quel campo (tutti i codici o tutti i cognomi ecc.) affiancati dalla posizione in cui sul file dati completo si trovano gli altri dati per quel record.

  L'indice è mantenuto in una forma che rende assai veloce trovare i valori che interessano (un certo codice o un certo cognome ecc.): ad esempio potrebbe essere ordinato (per sfruttare una ricerca dicotomica) o strutturato come un albero binario di ricerca; quest'ultima scelta è alla base della fortunatissima tecnica che sfrutta una forma modificata degli alberi binari, chiamata 'B+ tree'; essa è utilizzata per molte implementazioni del linguaggio Cobol, di molte librerie per la gestione degli archivi disponibili per gli altri linguaggi, e di alcuni famosi DBMS.

  Qualunque sia forma dell'indice, l'idea di base è che, grazie alla sua natura ordinata, saranno necessarie poche letture su di esso per localizzare il valore che interessa ed accedere all'indirizzo in cui si trova il resto dei dati nel file principale.

Arisi	4
Bini	2
Bruni	3
Calce	8
…	…

L'indice è ordinato per cognome. I dati di Arisi si trovano nel quarto record del file dei dati

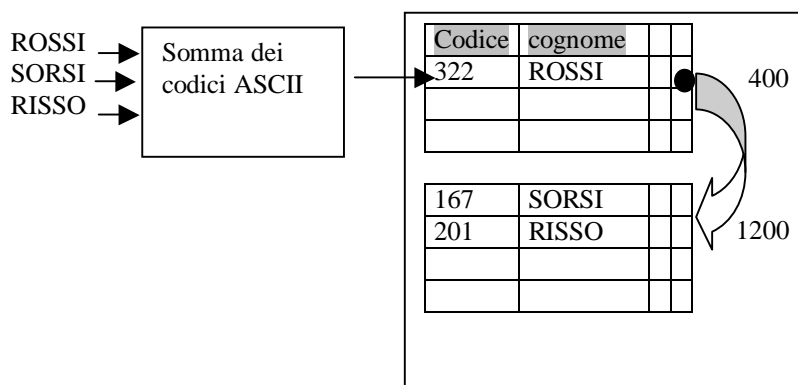
Calcestruzzi	Mario	V. Milano	Impiegato
Bini	Giorgio	V. Corta	Studente
Bruni	Massimo	V. Lunga	Studente
Arisi	Massimo	V. Larga	Avvocato

Il file dati principale non è ordinato rispetto ad alcun campo. L'accesso è mediato dall'uso dell'indice. Se l'accesso è random, ottenuta dall'indice la posizione dei dati di Arisi sarà sufficiente una sola lettura.

A titolo di esempio consideriamo un file di dati anagrafici contenenti un milione di record. Una ricerca dicotomica sul file indice nel caso più sfortunato richiederà 20 letture (20 =(circa) $\log_2(1000000)$ che è il numero di volte che può essere dimezzato un intervallo pari ad un milione) più una ventunesima nel file dati vero e proprio. Mediamente, in realtà, il numero di letture sarà minore di 20. Paragonate questo numero di letture con il numero medio di letture nel file dati principale disordinato (500000 !) per rendervi conto del guadagno… Qualcuno, giustamente, starà 'storcendo il naso per la puzza di bruciato': non sarebbe meglio tenere ordinato il file principale e sfruttare la ricerca dicotomica o qualsiasi altro algoritmo pensato per la gestione degli indici (evitando l'uso di questi ultimi)? In linea di principio questo sarebbe possibile ma molto più lento perché il file principale è di solito molto più grande dell'indice. Tenere aggiornato l'indice è invece un compito assai più leggero e, grazie a particolari strutturazioni quali quella prevista dalla tecnica 'B+ tree', può essere fatto praticamente in tempo reale senza rallentamenti avvertibili dall'utente. Inoltre è possibile avere più indici per accedere ai dati secondo diverse chiavi di ricerca (per codice, per cognome, per indirizzo ecc.): sarebbe veramente troppo oneroso riordinare il file principale in base al campo che di volta in volta interessa! Oppure mantenere tante copie del file dati quanti sono i criteri di ordinamento desiderati! Naturalmente tutto ha un limite: troppi indici tenderanno comunque a rallentare le operazioni a causa del sovraccarico necessario alla loro gestione (per ovviare in parte a questo problema nei moderni DBMS è possibile creare 'al volo' degli indici temporanei che verranno distrutti al termine dell'elaborazione al fine di non protrarre inutilmente l'aggravio della loro gestione).

- Relative: quando le unità informative (record) sono individuate da un numero che corrisponde alla loro posizione. E' così possibile, ad esempio, chiedere di inserire un record alla posizione 20 (anche se non sono state occupate tutte le precedenti) piuttosto che alla 5. Similmente è possibile chiedere di leggere un record ad una posizione qualsiasi. Tra i linguaggi che supportano questa organizzazione ricordiamo molti dialetti del BASIC e del COBOL.

- Hashing. Non sempre è conveniente applicarla. L'idea è che il processore impiega a fare un calcolo molto meno tempo di quello necessario a leggere dati dai supporti di massa: si tenta allora trovare una formula (legge di trasformazione) che, partendo da una chiave di accesso (ad esempio il cognome) calcoli la posizione del resto dei dati. Ad esempio potremmo stabilire che, dato un cognome di una persona, il resto dei dati si trova nel record alla posizione corrispondente al numero dei caratteri del cognome. In questo modo, volendo trovare i dati di 'Rossi' sapremmo in modo super rapido che si trovano sul quinto record del file (Rossi è una stringa di 5 caratteri). Ovviamente salta subito all'occhio il problema di questa formula troppo semplice: molti cognomi sono lunghi 5 caratteri e tutti i relativi record dovrebbero essere memorizzati nella stessa posizione … In effetti il problema dell'hashing sta proprio nella difficoltà di trovare una legge di trasformazione appropriata. Un altro esempio di legge: sommare i valori ASCII delle lettere corrispondenti ai cognomi. Anche questa legge genera posizioni uguali (sapresti dire perché e valutare se è migliore rispetto alla precedente?) … Purtroppo è impossibile trovare una legge di trasformazione perfetta. I problemi maggiori sono:

  conflitti di posizione: quando i valori dei campi di due record diversi generano lo stesso indirizzo; nell'esempio di prima tutti gli anagrammi di un cognome generano lo stesso indirizzo (vista la proprietà commutativa della somma); due tra le due tecniche più usate per aggirare questo problema sono o applicare in successione altre leggi di trasformazione per generare indirizzi diversi o gestire delle vere e proprie aree di 'overflow', come indicato nello schema seguente;

Ipotizziamo di dover inserire per primo il record di ROSSI; la legge di trasformazione generebbe l'indirizzo 400, usato per questo record; l'inserimento di SORSI richiederebbe di usare lo stesso indirizzo: trovandolo già occupato si memorizza il

nuovo record in un'area speciale (di overflow, cioè di traboccamento) che inizia all'indirizzo 1200 (valore scelto a caso); il 1200 viene memorizzato nel record di ROSSI per poter ritrovare in un secondo momento all'area di overflow. Per memorizzare RISSO si trova, ovviamente, già occupata la solita posizione 400. Si rileva (grazie al 1200 memorizzato prima) che esiste già un'area di owerflow. Quest'ultima viene fatta scorrere fino a supeare l'ultimo record che condivide con tutti gli altri lo stesso valore generato dalla legge di trasformazione. Se c'è ancora spazio viene aggiunto in coda il record di RISSO. Dovendo reperire le informazioni di RISSO sarà necessario scorrere l'area di overflow vanificando in parte il vantaggio dell'hashing, ma se le aree di overflow non sono esasperate è possibile ancora un notevole guadagno nei tempi complessivi.

<u>distribuzione ottimale dei valori generati</u>: immaginando di non ammettere omonimie (come potrebbe accadere per la descrizione dei prodotti in un magazzino) non è difficile inventarsi una legge di trasformazione hash che non causa conflitti; considerate questo algoritmo: facciamo corrispondere ogni lettera del cognome a numeri primi crescenti. Per ROSSI la R corrisponde al 2, la O al 3, la prima S al 5, la seconda S al 7 e la I all'11. Poi eleviamo ogni numero primo alla potenza corrispondente al codice ASCII delle lettere corrispondenti. Così eleveremo il 2 alla 82 (codice ASCII della R), il 5 alla 83 (codice ASCII della S), il 7 ancora alla 83 e così via. Poi moltiplichiamo tra loro i numeri ottenuti: il risultato è l'indirizzo richiesto. La matematica ci assicura (teorema dell'unicità della scomposizione in potenze di numeri primi di un numero) che due cognomi diversi genereranno due indirizzi diversi. Parrebbe di aver risolto il problema dei conflitti con un solo calcolo per qualsiasi campo da indicizzare di natura alfanumerica! Immaginate però di dover inserire il prodotto 'ZUCCA CON ZENZERO E ZAFFERANO'; il valore generato dall'algoritmo è MOLTO grande (MOLTO più grande del corrispondente decimale con tutti 9 !!). Il numero eccede anche le capacità di memorizzazione del più grande hd esistente, anche immaginando di associare un singolo byte su di esso ad un indirizzo diverso .... Potremmo scoprire, quindi, che per memorizzare anche solo il primo record dovremmo farlo ad un indirizzo impossibile da far corrispondere nella pratica: un miliardo di miliardi. Dovremmo creare tutti i record vuoti corrispondenti alle posizioni precedenti esaurendo (tempo permettendo) anche la batteria di hard disk più capiente dell'universo ...

Morale: la tecnica hash è utile solo in casi particolari da analizzare attentamente quando la velocità di reperimento dei dati è il fattore critico, diversamente l'organizzazione ad indici risulta più conveniente e sufficientemente performante.

OPERAZIONI DI BASE PER LA GESTIONE DEI FILE IN PASCAL

Qualunque sia l'organizzazione fisica possiamo individuare alcune operazioni di base tipiche. Per ciascuna di queste è importante, oltre al risultato, essere in grado di rilevare i motivi di eventuali errori (fine dello spazio su disco, disco in avaria, disco non trovato, file non trovato ecc.). Il controllo degli errori inevitabilmente rende il codice più pesante, più difficile da leggere: per questo motivo vedremo solo alcuni esempi seguendo i quali lo studente volenteroso (J) potrà perfezionare tutto il resto del codice.

NOTA: le considerazioni che seguono sono specifiche del linguaggio Pascal ma mantengono gran parte della loro validità anche per linguaggi diversi. Ogni volta differenzieremo tra file di testo e file tipizzati.

Collegamento al file esterno – procedura ASSIGN

Un file viene individuato dal suo nome e dal suo indirizzo fisico (path). Ad esempio C:\dati\2002\lettere\clienti\milano\ecc.\lettera.doc. In un programma Pascal per comandare le varie operazioni con un certo file si invece un nome logico scelto dal programmatore (ad esempio lettera). Il nome logico è più pratico e rende anche possibile cambiare il nome e la posizione di un file su disco senza dover necessariamente modificare il programma (ad esempio memorizzando i nomi e le posizioni fisiche dei file su un file a parte, questo sì fisso, che viene letto all'inizio del programma e con il cui contenuto vengono inizializzati tutti i nomi di file logici che si intendono usare).

Prima di poter usare un nome logico, che poi altro non è che una variabile di tipo speciale, è necessario indicare a quale file fisico lo si vuole far corrispondere.

FILE DI TESTO
Seguendo l'esempio di prima si vuole fa corrispondere il nome logico lettera al file fisico C:\dati\2002\lettere\clienti\milano\ecc.\lettera.doc. Si ricorre alla procedura standard Assign:

```
Assign( <variabile di tipo file> , <nome fisico del file> )
```

Applichiamola al nostro esempio:

```
var lettera: text;  (* lettera è il nome logico; il tipo text indica che si tratta di un file di testo *)
 ...
Assign(lettera, 'C:\ dati\2002\lettere\clienti\milano\ecc.\lettera.doc')
```

<u>Ora, e non prima, è possibile comandare le altre operazioni ...</u>

Possiamo rendere il nostro programma in grado di funzionare con diversi file fisici senza essere costretti a modificare il suo codice: è sufficiente indicare nell'assign una variabile stringa come nome fisico del file:

```
var lettera: text;  nomeFisico: string;
  ...
 writeln('Con quale file vuoi lavorare? ';
 readln(nomeFisico);
Assign(lettera, nomeFisico)
```

La procedura assign deve essere usata una volta sola prima di iniziare a lavorare con il file. Ovviamente dovremo comandare un assign per ciascun file che intendiamo usare.

FILE TIPIZZATI
Non c'è nessuna differenza nell'uso dell'assign. Dobbiamo però dichiarare il file in modo che sia riconosciuto come tipizzato:

```
 type

  TPersona=record
    cognome: string[30];  (* e' obbligatorio definire il numero dei caratteri della stringa !  *)
    codice: integer;
  end;

  fpersona=file of TPersona; (* senza il tipo non potremmo passare un file come parametro ... *)

var
    elenco: fPersona;
 ...
assign(elenco, 'C:\dati.dat')
```

Creazione – procedura REWRITE

L'assegnazione da sola non basta per usare un file. E' necessario crearlo. Un file appena creato risulterà vuoto e non si è obbligati ad inserire subito nuovi dati (anche se di solito lo si fa). Un file creato e lasciato vuoto risulterà con 0 byte nelle cartelle di windows. Non confondete quindi la creazione con l'inserimento dei dati.

La creazione può fallire per vari motivi:
- il percorso/nome del file fornito dal programmatore non è valido
- il supporto di massa non è disponibile (nastro non montato, floppy non inserito, CD non scrivibile)
- il supporto è in avaria (floppy o hard disk rotti ...)
- spazio sul supporto esaurito (anche se è difficile che non ci sia spazio almeno per creare il file vuoto)

Ma vediamo la sintassi dell'istruzione di creazione (identica per file di testo e tipizzati):

```
rewrite(nome_file_logico)
```

Quindi, continuando gli esempi dell'istruzione precedente (assign) senza ripetere la dichiarazione dei tipi e delle variabili:

TESTO	TIPIZZATI	
assign(lettera, '...');	assign(elenco);	come già detto, la sintassi è identica
rewrite(lettera)	rewrite(elenco);	

ATTENZIONE!!!

Se comandate una rewrite su un file già presente in quella posizione e con lo stesso nome, quest'ultimo verrà eliminato, e si ricomincerà con un file vuoto. Più avanti vi verrà spiegata una tecnica per verificare la presenza di un vecchio file senza distruggerlo e chiedere conferma prima di procedere con la rewrite.

Chiusura – procedura CLOSE

Ora il file esiste (vuoto) ed il programmatore potrebbe continuare con le istruzioni che aggiungono righe/schede. Oppure terminare subito le operazioni. In ogni caso il termine delle operazioni con un certo file va segnalato con il comando CLOSE: close(nome_file_logico). Di nuovo, non ci sono differenze tra file di testo e tipizzati.

TESTO	TIPIZZATI
assign(lettera, '...');	assign(elenco);
rewrite(lettera)	rewrite(elenco);
close(lettera)	close(elenco)

Questa è ovviamente la sequenza dei comandi nel caso il programmatore decidesse di creare i file senza registrare alcun dato su di essi (ovviamente potrà farlo in seguito, 'riaprendo' i file). Diversamente immaginate tra la rewrite e la close le istruzioni che aggiungono nuove righe (file di testo) o schede (file tipizzati).

N OTA: close restituisce anche al sistema operativo tutte le risorse usate per la gestione del file.

Scrittura – le procedure APPEND (file di testo), WRITELN (file di testo) e WRITE (file tipizzati)

Le procedure writeln e write permettono rispettivamente di aggiungere una riga ad un file di testo o un record ad un file tipizzato.

Con il Pascal, per i file di testo ed i tipizzati gli inserimenti possono essere effettuati solo in fondo al file; non c'è modo di inserire nuove informazioni in mezzo al file senza crearne uno nuovo.

L'operazione di inserimento potrebbe fallire:
- lo spazio sull'unità è esaurito
- l'unità non è disponibile (ad esempio floppy non inserito)
- l'unità è andata in avaria
- il file è già stato aperto da un utente in modalità che impedisce gli inserimenti agli altri utenti

FILE DI TESTO
Ad un file di testo possono essere esclusivamente aggiunte righe in fondo al file. Non è possibile, ad esempio, posizionarsi in una posizione intermedia ed inserire una riga. Il file deve essere stato prima predisposto o con rewrite (creando un file vuoto siamo ovviamente in fondo al file ...) o con la procedura append(nome_file_logico).
Quest'ultima deve essere usata quando si vogliono fare aggiunte ad un file di testo che contiene già alcune righe.

Senza l'append dovremmo sempre creare da capo i file di testo riscrivendo anche le vecchie righe (vi ricordo che rewrite è distruttiva!).

L'istruzione che aggiunge le righe al file è una variante dell'arcinota writeln. Semplicemente dovremo indicare il nome del file logico su cui scrivere e la stringa che rappresenta la riga da aggiungere: writeln(nome_file_logico, stringa).

Dopo ogni writeln sul file, il punto di inserimento è automaticamente fatto avanzare.

Come al solito, gli esempi che seguono fanno riferimento alle dichiarazioni dei paragrafi precedenti.

File creato e scritto immediatamente

```
Var riga: string;

begin
Assign(lettera,'c:\...');
rewrite(lettera);  (* FILE AZZERATO! *)

repeat
  writeln('Inserire la riga da scrivere sul file');
  readln(riga);

  if riga<>'FINE' then
    writeln(lettera, riga);

until riga='FINE';
close(lettera);
end.
```

Aggiunga di righe ad un file esistente

```
Var riga: string;begin

begin
Assign(lettera,'c:\...');
append(lettera);  (* VECCHIE RIGHE CONSERVATE!*)

repeat
  writeln('Inserire la riga da scrivere sul file');
  readln(riga);

  if riga<>'FINE' then
    writeln(lettera, riga);

until riga='FINE';
close(lettera);
end
```

In entrambi gli esempi chi usa il programma comunica che le righe da aggiungere sul file di testo sono terminate digitando la stringa ‘FINE’.

FILE TIPIZZATI

E’ molto importante notare che una scheda (record) viene letta/scritta tutta insieme, non un campo alla volta. Per le operazioni deve essere dichiarata una variabile record dello stesso tipo delle schede che si vogliono leggere/scrivere.

Per aggiungere una scheda dovremo prima memorizzare i dati nella variabile record e poi comandare la registrazione di quest’ultima nel file tipizzato.

L’istruzione che materialmente aggiunge la scheda è write(nome_file_logico, variabile_record_con_dati).

Dopo ogni write sul file, il punto di inserimento è automaticamente fatto avanzare.

File creato e scritto immediatamente

```
type

  TPersona=record
    cognome: string[30];  (* [30]: ricordatevi il numero di caratteri! *)
    codice: integer;
  end;

var
    elenco: file of TPersona; (* la variabile file; si sarebbe anche potuto definire prima un tipo ‘file of TPersona’ *)
    unaPersona: TPersona;

begin
  assign(elenco,'c:\prova.dat');

  (* crea il file azzerandolo, pronto per la prima scheda *)
  rewrite(elenco);

repeat
  writeln('Inserimento da tastiera di un record da trasferire poi sul file');

  write('Inserire un codice: (zero per finire)' );
  readln(unaPersona.codice);
  write('Inserire un cognome: ');
  readln(unaPersona.cognome);

  if unaPersona.codice<>0 then
    write(elenco,unaPersona);   AGGIUNGE LA SCHEDA IN FONDO AL FILE ED AVANZA

until unaPersona.codice=0;

close(elenco);
end.
```

Chi usa il programma comunica che le schede da aggiungere sono terminate digitando come codice il valore zero.

<u>Aggiunga di righe ad un file esistente – PROCEDURA SEEK E FUNZIONE FILESIZE</u>

L'append non esiste ma può essere simulata spostandosi in fondo al file con un comando speciale. Un grosso vantaggio, infatti, dei file tipizzati è la possibilità di spostarsi su una scheda qualsiasi senza dover leggere quelle che la precedono. Il comando è seek(nome_file_logico, posizione). ATTENZIONE: si inizia a contare da zero!! Quindi la posizione del primo record è la zero, quella del secondo record è due e così via. Per spostarsi sul primo record allora si userà seek(file , 0), per spostarsi invece sul decimo seek(file, 9).

Per simulare l'append dovremmo allora spostarci oltre l'ultimo record. Se ci fossero N record l'istruzione sarebbe seek(file, N): infatti seek(file, N-1) individuerebbe proprio l'N-mo record ed indicando N come posizione ci posizioneremmo sul record N+1 (cioè sul nuovo che abbiamo intenzione di inserire), sempre in virtù del fatto che si conta da 0. Già, ma come si fa a sapere quanti record ci sono nel file? Un metodo brutale potrebbe essere quello di leggerli tutti fino a raggiungere la fine del file (EOF) ma sarebbe molto dispendioso. Per fortuna esiste un comando specifico (solo per i file tipizzati; secondo voi perché ?): filesize(nome_file_logico).

Inoltre, prima di poter usare seek il file deve essere predisposto all'uso (aperto) con la procedura reset(nome_file_logico), da usare ovviamente dopo l'assign.

I comandi che portano oltre la fine del file sono allora:

```
reset(nome_file_logico);  QUESTO COMANDO PREDISPONE IL FILE ALL'USO, LO 'APRE'
seek( nome_file_logico, filesize(nome_file_logico) )
```

Ed ecco un esempio completo:

```
type
  TPersona=record
    cognome: string[30];  (* e' obbligatorio definire il numero dei caratteri della stringa !  *)
    codice: integer;
  end;

  fpersona=file of TPersona; (* senza il tipo non potremmo passare un file come parametro ... *)

var
    elenco: fPersona;
 ...
assign(elenco, 'C:\dati.dat');

reset(elenco);
seek(elenco, filesize(elenco ); SPOSTATI OLTRE L'ULTIMA SCHEDA PRESENTE NEL FILE

repeat
  writeln('Inserimento da tastiera di un record da trasferire poi sul file');

  write('Inserire un codice: (zero per finire)' );
  readln(unaPersona.codice);
 write('Inserire un cognome: ');
 readln(unaPersona.cognome);

if unaPersona.codice<>0 then
  write(elenco,unaPersona); LA SCHEDA IN FONDO AL FILE ED AVANZA

until unaPersona.codice=0;

close(elenco);
```

Lettura – la procedura RESET, READLN (file di testo), READ (file tipizzati)

Per usare un file già creato in precedenza senza distruggerlo lo si deve prima predisporre all'uso con la procedura reset(nome_file_logico).

Con reset in pratica si notifica l'uso del file al sistema operativo predispone le risorse necessarie (area di memoria RAM per i trasferimenti da e verso i dischi, tabelle per tenere traccia delle operazioni ecc.). Queste strutture occupano memoria centrale preziosa, per cui bisognerebbe tenere 'aperti' solo i file necessari.

Quasi tutti i linguaggi distinguono tra diversi tipi di apertura (Cobol, 'C') a seconda delle operazioni di I/O che si intendono fare: sola lettura, solo scrittura, lettura / scrittura, aggiunta a fine file (append). E' impossibile in questa sede discutere tutte le varianti: consultate attentamente il manuale del linguaggio in uso.

Anche l'apertura può fallire:
- il percorso/nome del file non è valido
  - il supporto di massa non è disponibile (nastro non montato, floppy non inserito, CD non scrivibile)
  - il supporto è in avaria (floppy o hard disk rotti ...)
  - il file è già stato aperto da un utente in modalità esclusiva (blocca i tentativi di accesso di tutti gli altri)

Con il Pascal, dopo l'apertura con reset di un file di testo si è pronti per leggere la prima riga. Dopo l'apertura di un file tipizzato si è pronti per leggere il primo record. Una funzione fondamentale per una lettura corretta sia dei file di testo che di quelli tipizzati è la funzione EOF(nome_file_logico): che restituisce TRUE se siamo alla fine del file, FALSE altrimenti.

Vediamo alcuni esempi concreti di lettura.

FILE DI TESTO
```
... dichiarazioni ...
    (* riapriamo il file per lettura *)
    reset(lettera);

    (* leggiamo tutte le righe fino alla fine *)
    while not eof(lettera) do
    begin
      readln(lettera,riga);
      writeln(riga)
    end;
    close(lettera);
```

FILE TIPIZZATI
```
... dichiarazioni ...
    (* riapriamo il file per lettura *)
    reset(elenco);

    (* leggiamo tutte le schede fino alla fine *)
    while not eof(elenco) do
    begin
      readln(elenco,unaPersona);
      writeln(unaPersona.cognome,' - ',unaPersona.codice)
    end;
    close(lettera);
```

Non perdiamo la bussola – puntatore all'area di lavoro, BOF ed EOF

Nell'uso di un file è sempre molto importante rendersi conto in che punto dello stesso si sta lavorando. Possiamo immaginare un 'puntatore' che indica la posizione:

file appena creato; è vuoto; siamo all'inizio del file (BOF, Begin Of File) e contemporaneamente alla fine (EOF, End Of File)

il file contiene già dei dati; siamo posizionati in un punto intermedio per leggere un blocco di dati;

<u>solo con i file tipizzati</u> potremmo anche riscrivere (aggiornare) il blocco dei dati

siamo alla fine del file; in questa posizione possiamo aggiungere nuovi blocchi di dati

NOTA BENE

Tutte le operazioni di lettura/scrittura spostano sempre il puntatore in avanti di una 'posizione' (una riga per i file di testo, una scheda per i file tipizzati) verso la fine del file (EOF, End Of File).

Dopo la creazione siamo all'inizio, ma anche alla fine (file è vuoto!) e quindi pronti per aggiungere nuovi blocchi di dati. Ad ogni inserimento di una riga/scheda la posizione viene spostata automaticamente oltre la fine del file e si è pronti per aggiungere un nuovo blocco di dati.

Dopo aver letto una riga/scheda (se il file già ne contiene) la posizione si sposta automaticamente in avanti e si è pronti per leggere la successiva.

Aggiornamento (QUESTA OPERAZIONE E' POSSIBILE SOLO PER I FILE TIPIZZATI)

Il file deve esistere e deve essere prima aperto con reset; si deve localizzare la scheda (record) da aggiornare cioè posizionarsi in corrispondenza del punto a partire dal quale è necessario sovrascrivere le vecchie informazioni con quelle nuove.

Poi si legge la vecchia scheda, per due motivi:

- recuperare quelle informazioni della scheda che non devono cambiare per poterle riscrivere uguali al momento della riscrittura della scheda; diversamente dovremmo costringere chi usa il programma a reinserirle (improponibile); ad esempio in una scheda anagrafica potrebbe cambiare solo l'indirizzo: la lettura della vecchia scheda recupera tutte le altre informazioni (cognome, nome, codice, ecc.) che verranno riscritte uguali insieme al nuovo indirizzo;

- poter proporre le vecchie informazioni sul video facilitando chi usa il programma nell'inserimento dei campi del record che devono essere cambiati;

La lettura della vecchia scheda fa però avanzare al record successivo e non siamo perciò più posizionati nel punto giusto per l'aggiornamento. Se scrivessimo in questa situazione andremmo ad aggiornare in realtà la scheda successiva (od aggiungeremmo una nuova scheda se fossimo avanzati oltre la fine del file). E' quindi necessario ripetere il posizionamento con seek prima di riscrivere la scheda.

L'aggiornamento può fallire:
- i parametri indicati sono errati
- l'unità non è disponibile (ad esempio il floppy è stato nel frattempo tolto)
- l'unità è andata in avaria

Dopo l'aggiornamento ci si trova poi all'inizio dell'unità informativa successiva.

Ecco un esempio che propone la sequenza delle operazioni per modificare solo il cognome della persona corrispondente al secondo record (posizione 1, ricordatevi che si conta da zero ...):

```
type
  TPersona=record
    cognome: string[30];  (* e' obbligatorio definire il numero dei caratteri della stringa !  *)
    codice: integer;
  end;

  fpersona=file of TPersona; (* senza il tipo non potremmo passare un file come parametro ... *)

var
    elenco: fPersona;
 ...
assign(elenco, 'C:\dati.dat'); reset(elenco);

(* ora modifichiamo  il secondo record  la sequenza standard e':
  - seek alla posizione della scheda da modificare
  - lettura della scheda con vecchi dati

    ATTENZIONE: a questo punto abbiamo superato la scheda da aggiornare

  - acquisizione nuovi dati che vengono sostituiti a quelli
    vecchi nella scheda

  - torniamo indietro alla posizione giusta per scrittura ripetendo
    la seek iniziale (può anche essere fatta subito dopo la lettura, per non dimenticarsene ...)

  - scrittura su file (aggiornamento) della scheda con i dati modificati *)

 seek(elenco,1);                    (* seek prima della scheda da modificare *)
 read(elenco, unaPersona);          (* lettura scheda con vecchi dati *)

 writeln('-------- VECCHI DATI -----');
 writeln(unaPersona.cognome,' - ',unaPersona.codice);

 write('Inserire il nuovo cognome: ');
 readln(unaPersona.cognome);

 (* riposizionamoci nel posto giusto per la scrittura *)
 seek(elenco,1);


 (* ora la variabile record contiene il vecchio codice ed il nuovo cognome: riscriviamola sul file *)
 write(elenco,unaPersona);
```

Eliminazione

FILE DI TESTO
Non è possibile eliminare una riga da un file di testo. L'unica possibilità è quella di ricopiare tutte le righe meno quelle da cancellare in un nuovo file (operazione assai dispendiosa).

FILE TIPIZZATI
Di nuovo, non è possibile eliminare fisicamente un record. Però, grazie alla possibilità di riscrittura si può'contrassegnare' una scheda in qualche modo convenzionale scelto dal programmatore. Ad esempio potrebbe esserci un campo 'codice' e decidere che se contiene zero la scheda è da considerarsi eliminata. Di tanto in tanto potremmo comandare in momenti di basso utilizzo dell'archivio (di notte, ad esempio) una riorganizzazione ricopiando in un nuovo file le schede non 'eliminate'.

Come controllare gli esiti delle operazioni richieste (GESTIONE DEGLI ERRORI)

Il sistema operativo, sollecitato da comandi visti, può dare conferma in merito al loro soddisfacimento oppure segnalare un errore. I casi che portano al fallimento possono essere vari: spazio sul supporto esaurito, il supporto indicato non è scrivibile (CD ROM), il supporto o un altro dispositivo necessario per il suo uso (ad es. il controller dell'hd) è in avaria, le risorse di sistema non consentono al momento l'operazione (troppi programmi in uso, troppi utenti collegati, troppi file in uso, collegamento ad Internet al momento interrotto ecc.), il programma (e di conseguenza l'utente che lo ha eseguito) non ha l'autorizzazione necessaria, il percorso indicato nel nome del file non esiste o non è corretto, il nome del file indicato non è corretto ecc.

La gestione di queste situazioni può avvenire fondamentalmente in due modi:

- Il sistema operativo rileva le anomalie ed agisce automaticamente: spesso questo significa un messaggio sullo schermo e la terminazione forzata del programma che ha generato l'errore. In qualche caso, ma con uno sforzo notevole, il programmatore può sostituire all'azione standard del sistema operativo le sue 'routines' di gestione degli 'interrupt' relativi (avete imparato a vostre spese come questo sia difficile da realizzare, soprattutto in 'assembly)'.

- Il programma in esecuzione riceve una notifica di errore, dopo un'operazione tentata, ed ha la possibilità di gestirlo e di ripristinare il funzionamento del programma in modo 'accettabile'. La notifica può avvenire tramite:

  un valore restituito dalla funzione di I/O tentata: ad esempio la 'fopen' del 'C' dovrebbe restituire un puntatore ad un file (una struttura a record usata per le operazione di I/O con il corrispondente file fisico) ma potrebbe restituire 'null' in caso di insuccesso;

  un controllo esplicito del programmatore dopo aver tentato un'operazione: ad esempio in Pascal è possibile chiamare la funzione IOResult: se il valore restituito è 0 allora l'operazione è andata a buon fine, diversamente il valore indica il tipo di errore (ad esempio 1=file non trovato, 2=disco pieno ecc.)

  l'uso di una struttura di controllo ad alto livello specifica: gli ambienti di programmazione più avanzati prevedono potenti strutture di programmazione per la gestione degli errori; ad esempio l'Object Pascal offre il costrutto 'try … except …' in cui dopo il termine 'try' (tenta) si mettono le istruzioni che potenzialmente potrebbero causare errori; nella sezione 'except', quasi come in un 'case' si elencano, usando dei nomi significativi, gli errori che si intendono 'intercettare' ed i relativi sottoprogrammi scritti dal programmatore per gestirli; molti dialetti del Basic (tra cui Visual Basic e Visual Basic for Application, VBA, il linguaggio degli applicativi Office) offrono un costrutto analogo: 'on error ….'

ESEMPIO COMPLETO 1

L'esempio seguente mostra come procedere alla creazione di un file controllando prima se ne esiste già uno vecchio.

Se esiste l'operazione di *reset* va a buon fine e possiamo accorgercene dal valore 0 restituito dalla funzione *IOResult*: in questo caso si chiede conferma a chi sta usando il programma prima di procedere con una *rewrite* distruttiva.

Se non esiste l'operazione di *reset* fallisce e possiamo accorgercene dal valore diverso da 0 restituito dalla funzione *IOResult*: in questo caso si può procedere con una *rewrite* distruttiva senza chiedere conferma.

```
type
  TPersona=record
    cognome: string[30];  (* e' obbligatorio definire il numero dei caratteri della stringa !  *)
    codice: integer;
  end;

  fpersona=file of TPersona; (* senza il tipo non potremmo passare un file come parametro … *)

var
    elenco: fPersona;

begin

(* la riga che segue abilita alla gestione degli errori e va inserita così com'è su una riga isolata;
se la vostra tastiera non ha le graffe vi ricordo che potete tenere premuto ALT di sinistra e digitare 123 o 125 sul
tastierino numerico con tasto 'Bloc Num' del tastierino attivato *)

{$I-}

assign(elenco,'c:\prova.dat');
reset(elenco);
risposta:='SI';

if IOResult=0 then
  begin
    writeln('Il file esiste gia'': procedo? (SI/NO)');
    readln(risposta);
    close(elenco)
  end;

if risposta='SI' then
begin
  rewrite(elenco);
 ecc. ecc.
close(elenco)
end;

end.
```

NOTA. Nel caso volessimo fare un aggiornamento o una lettura da un file la logica sarebbe semplicemente invertita: se non si riesce ad aprire il file ci si ferma; se ci si riesce si procede …

IOResult fornisce codici numerici diversi a seconda delle situazioni e volendo essere più precisi potremmo usare un *case*:

```
case IOResult of
   1: ….
   2: ….
End.
```

Ecco i codici più comuni (sequenza completa nell'help in linea del turbo pascal cercando 'run time error codes')

2	File not found
3	Path not found
15	Invalid drive number
100	Disk read error
101	Disk write error
102	File not assigned
103	File not open
104	File not open for input
105	File not open for output
150	Disk is write-protected
152	Drive not ready
156	Disk seek error
157	Unknown media type
158	Sector Not Found
162	Hardware failure

ESEMPIO COMPLETO 2

In questo secondo esempio gestiamo una semplice scheda di dati anagrafici; per sfruttare al meglio l'istruzione seek immaginiamo che ogni scheda sia individuata da un codice numerico che facciamo corrispondere alla sua posizione come record. Quindi se si cerca la scheda con codice 7 si utilizza seek(file, 6) e così via. Il codice viene mantenuto strettamente progressivo ed è assegnato in automatico dal programma

Questo esempio fa uso di procedure e funzioni con parametri. In alcuni casi fa anche il controllo degli errori con i file.

Il programma tiene sempre sotto controllo il numero dei record presenti nell'archivio (per poter generare in automatico il codice da assegnare alle schede). In partenza, infatti, se il file è vuoto mette la variabile inseriti a zero altrimenti gli assegna il numero di record presenti nel file. Il programma tiene poi aggiornato questo contatore ad ogni inserimento o cancellazione. La cancellazione non avviene in modo fisico ma mettendo a FALSE un boolean del record (campo attivo).

Le scelte di chi usa il programma sono gestite da una funzione menu che riceve l'elenco delle voci da presentare (un vettore di stringhe) e restituisce il valore della scelta fatta.

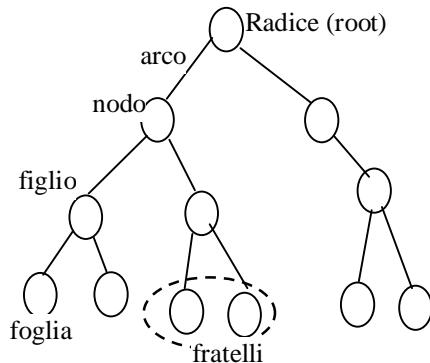
```
program ProvaFileTipizzati;

(* e' necessario disabilitare l'intercettazione degli errori di I/O da
   parte del Turbo Pascal al fine di gestirli in modo personalizzato *)

{$I-}  (* non cancellare ! *)

const MAX_VOCI_MENU=9;
type
  voci_menu=array[1..MAX_VOCI_MENU] of string;

  TPersona=record
    codice: integer;
    cognome: string[30];
    attivo: boolean; (* false: record da considerare cancellato *)
  end;

  fDati=file of TPersona;

var
    dati: fDati; i,scelta:integer; ultimo_codice,unCodice: longint;
    nomeFile,riga: string;   unaPersona: TPersona;
    procedi: boolean; conferma: char;
    menu_database: voci_
menu;

(* restituisce true se il file il cui nome logico è passato come parametro esiste, false altrimenti *)
function esiste(var f: fdati): boolean;
var esito: boolean;
begin
  reset(f);

  if IOResult=0 then
  begin
    close(f);
    esito:=true
  end
  else
    esito:=false;
  esiste:=esito
end;
```

```
(* riceve un vettore di stringhe ed il numero di queste da considerare e presentare come menu sullo schermo;
intitola anche il menu con un altro parametro stringa; restituisce il numero della voce del menu scelta *)
function menu(voci: voci_menu; n_voci: integer;titolo: string): integer;
var i,voce_scelta: integer;
begin
  repeat
    writeln(titolo);writeln;
    for i:=1 to n_voci do
      writeln(voci[i]);
    write('Scegli -> ');
    readln(voce_scelta);
    if (voce_scelta<1) or (voce_scelta>n_voci) then
    begin
      writeln('Scelta errata!! (UN TASTO PER RIPROVARE)');
      readln
    end
  until (voce_scelta>0) and (voce_scelta<=n_voci);
  menu:=voce_scelta
end;

procedure visualizza_file(var f: fDati);
var persona: TPersona;
begin
  reset(f);

  if ioResult=0 then  (* se riesco ad aprirlo significa che non è vuoto ... *)
  begin
    (* leggiamo tutti i record fino alla fine *)
    while not eof(f) do
    begin
      read(f,persona);
      if persona.attivo then (* salto i cancellati ... *)
        writeln(persona.codice, ' - ',persona.cognome)
    end;
    close(f)
  end
end;

begin  (* PROGRAMMA PRINCIPALE *)
  (* clrscr; *)
  nomeFile:='c:\anagrafe.dat';
  assign(dati,nomeFile);
  ultimo_codice:=0;

  if esiste(dati) then
  begin
    reset(dati);
    ultimo_codice:=filesize(dati);  (* filesize restituisce il n. di record *)
  end
  else
  begin
    rewrite(dati);
    ultimo_codice:=0
  end;
  close(dati);


  (* preparo le voci del menu prima di usarlo *)
  menu_database[1]:='1 - Azzera archivio';
  menu_database[2]:='2 - Inserimento nuovi nominativi';
  menu_database[3]:='3 - Ricerca dati per codice';
  menu_database[4]:='4 - Modifica dati';
  menu_database[5]:='5 - Elimina nominativo';
  menu_database[6]:='6 - Visualizza Archivio';
  menu_database[7]:='7 - Fine Operazioni';
```

```
repeat
  scelta:=menu(menu_database,7,'Gestione Nominativi');
  case scelta of
    1:begin  (* azzeramento *)
        if ultimo_codice>0 then
        begin
          writeln('Attualmente sono inseriti ',ultimo_codice, ' nominativi; confermi distruzione? (s/n)');
          readln(conferma);
          if (conferma='s') or (conferma='S') then
          begin
            rewrite(dati);
            ultimo_codice:=0
          end
        end;
        writeln('Fatto!! INVIO per continuare');
        readln
      end;
    2:begin  (* inserimento *)
        inc(ultimo_codice);
        with unaPersona do
        begin
          codice:=ultimo_codice;
          write('Inserire cognome: ');
          readln(cognome);
          attivo:=true
        end;
        reset(dati);
        seek(dati, FileSize(dati)); (* append ... *)
        write(dati,unaPersona);
        close(dati)
      end;
    3:begin  (* ricerca *)
        if ultimo_codice>0 then
        begin
          write('Inserire il codice che interessa: ');readln(unCodice);
          if unCodice<=ultimo_codice then
          begin
            reset(dati);
            seek(dati,unCodice-1); (* i record sono numerati a partire da 0 *)
            read(dati,unaPersona);
            if unaPersona.attivo then (* salto i cancellati ... *)
              writeln(unaPersona.codice, ' - ',unaPersona.cognome)
            else
              writeln('Nominativo cancellato')
          end
          else
            writeln('Ci sono solo ',ultimo_codice,' nominativi!')
        end
        else
          writeln('File Vuoto');

        writeln('INVIO per continuare');
        readln;

      end;
    4:begin  (* modifica *)
        if ultimo_codice>0 then
        begin
          write('Inserire il codice che interessa: ');readln(unCodice);
          if unCodice<=ultimo_codice then
          begin
            reset(dati);
            seek(dati,unCodice-1); (* i record sono numerati a partire da 0 *)
            read(dati, unaPersona);
            if unaPersona.attivo then (* salto i cancellati ... *)
```

```
              begin
                writeln('Vecchi dati: ',unaPersona.codice, ' - ',unaPersona.cognome);
                write('Inserire nuovo cognome: ');
                readln(unaPersona.cognome);
                (* per riscrivere devo rimettermi PRIMA del record *)
                seek(dati,unCodice-1);
                write(dati,unaPersona);
                writeln('Fatto!')
              end
              else
                writeln('Nominativo cancellato')
            end
            else
              writeln('Ci sono solo ',ultimo_codice,' nominativi!')
          end
          else
            writeln('File Vuoto');

          writeln('INVIO per continuare');
          readln
        end;
      5:begin   (* cancellazione *)
          if ultimo_codice>0 then
          begin
            write('Inserire il codice che interessa: ');readln(unCodice);
            if unCodice<=ultimo_codice then
            begin
              reset(dati);
              seek(dati,unCodice-1); (* i record sono numerati a partire da 0 *)
              read(dati, unaPersona);
              (* dovrei chiedere conferma, ma per brevità ... *)
              if unaPersona.attivo then (* salto i cancellati ... *)
              begin
                seek(dati,unCodice-1);
                unaPersona.attivo:=false;
                write(dati,unaPersona);
                writeln('Fatto!')
              end
              else
                writeln('Nominativo gia'' cancellato')
            end
            else
              writeln('Ci sono solo ',ultimo_codice,' nominativi!')
          end
          else
            writeln('File Vuoto');

          writeln('INVIO per continuare');
          readln
        end;
      6: begin (* visualizzazione *)
           if ultimo_codice>0 then
             visualizza_file(dati)
           else
             writeln('File Vuoto');

          writeln('INVIO per continuare');
          readln
         end
    end
  until scelta=7;

end.
```

RICORSIONE

Esistono problemi tutto sommato semplici da descrivere che portano ad algoritmi incredibilmente complicati se trattati con gli strumenti di programmazione visti sino ad ora… Ad esempio: molte situazioni reali sono adatte per essere rappresentate con una struttura chiamata *albero*. Un albero è formato da elementi chiamati *nodi* collegati da *archi:*

Nel disegno i *nodi* sono rappresentati dai cerchietti, gli *archi* dai segmenti che li congiungono. In realtà si tratta di un caso particolare di albero chiamato *albero binario*, in quanto ogni nodo può avere al massimo due figli (ma anche uno o nessuno). Gli alberi più complessi (*generici*, senza limiti sul numero di figli) possono comunque essere rappresentati con un albero binario costruito in modo 'furbo' (scoprirete più avanti nel corso come …).
Il nodo iniziale, quello in cima per intenderci, è chiamato *root* (radice). I nodi terminali (quelli senza figli) sono chiamati *foglie*. I figli diretti di uno stesso nodo sono *fratelli* tra loro.

Qualche esempio d'uso: in un gioco un albero può rappresentare per ogni possibile 'mossa' le risposte dell'avversario e per ciascuna di queste le possibili contromosse e così via; i sistemi operativi utilizzano strutture ad albero per memorizzare la struttura delle cartelle e relative sotto cartelle; un albero genealogico; l'organigramma delle figure di un'azienda (boss, capi reparto, dipendenti ecc.); alcune delle tecniche più sofisticate di gestione degli archivi usano strutture ad albero (B+ tree).

E' ovvio che ciò che conta è il 'contenuto' di ogni nodo. Per un gioco (scacchi ad esempio) ogni nodo potrebbe memorizzare la situazione appena prima o appena dopo una certa mossa (la matrice della scacchiera). Il sistema operativo potrebbe memorizzare l'elenco dei file in quella directory (non è proprio così …); per l'albero genealogico i dati anagrafici della persona associata a ciascun nodo ecc.

Ma come si rappresentano gli archi? La soluzione classica prevede la gestione tramite memoria dinamica: ogni nodo memorizza i puntatori ai figli. Ad esempio:

```
type
  Pun=^Nodo;
  Nodo=record
     ParteInformativa: string;
     sx: Pun; (* memorizza il puntatore al figlio di sinistra *)
     dx: Pun   (* memorizza il puntatore al figlio di destra *)
   end
```

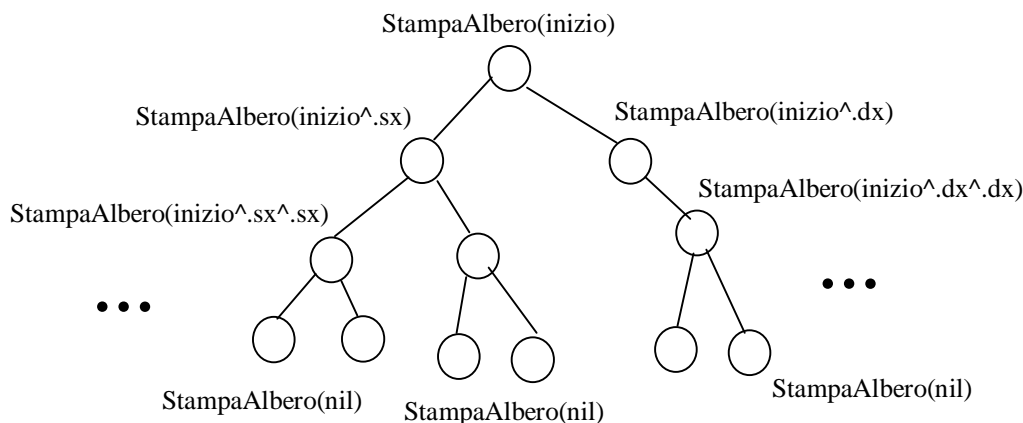
OK, immaginiamo di avere l'albero già perfettamente costruito e che la variabile puntatore *inizio* punti al primo nodo (*root*). Ed ecco la sfida: scrivere un programma che stampi l'elenco di tutte le parti informative memorizzate nell'albero. E' una richiesta banale: se non riuscissimo a fare almeno questo l'albero sarebbe inutilizzabile! Da un punto di vista logico si tratta di *visitare* tutti i nodi e per ciascuno stampare la parte informativa. Sfido chiunque a trovare una soluzione che faccia uso di cicli: essa esiste ma è di **elevata** complessità. E' anche quasi incomprensibile … Ed è pure lunga alcune decine di righe. Guardate invece l'eleganza, la semplicità e l'estrema leggibilità di questa soluzione (**ricorsiva**):

```
procedure StampaAlbero(nodo: pun);
begin
  if nodo<>NIL then
  begin
     1.  writeln(nodo^.ParteInformativa);
     2.  StampaAlbero(nodo^.sx);
     3.  StampaAlbero(nodo^.dx)
 end
end;
```

NOTA: ho numerato le righe per facilitare il successivo commento del codice.

Autore: Fabrizio Camuso (email: camuso@camuso.it sito web: www.camuso.it)

Considerando l'if' sono solo sei righe !!! Sembra quasi impossibile… La stampa completa è comandata con *StampaAlbero(inizio)* dove *inizio* è il puntatore al primo nodo dell'albero (la radice). La logica di esecuzione è assai semplice: stampare la parte informativa del nodo di cui si è ricevuto il puntatore e chiamare quindi prima la stampa pe il figlio di sinistra (*StampaAlbero(nodo^.sx)*) e poi per quello di destra (*StampaAlbero(nodo^.dx)*). Il meccanismo si ripete sui rispettivi figli sinistra/destra fino a raggiungere le foglie.

La procedura chiama quindi sé stessa (ricorre ai servizi di sé stessa, da cui il termine ***ricorsione***) ma ogni volta passando un puntatore che la fa avvicinare alle foglie. Con queste ultime verranno chiamate delle 'StampaAlbero' con parametro uguale a NIL (le foglie non hanno nodi sotto di esse e quindi hanno il valore NIL nei loro puntatori *sx* e *dx*) terminando la catena delle chiamate.

E' molto importante capire che in un certo istante avremo più copie della procedura *StampaAlbero* attivate (ma una sola 'funzionante'). Infatti la prima chiamata (StampaAlbero(inizio)) chiama sé stessa con il puntatore al suo figlio di sinistra e deve poi letteralmente rimanere in attesa alla riga n. 2 aspettando la fine dell'esecuzione di *StampaAlbero(inizio^.sx)*. E quest'ultima dopo aver stampato la parte informativa del suo nodo aspetterà alla stessa riga per lo stesso motivo e così via. Riferendosi al disegno precedente, verranno chiamate in sequenza ben 4 *StampaAlbero* che esplorano tutto la parte a sinistra dell'abero, fino a raggiungere la prima foglia (l'ultimo nodo in basso a sinistra). Solo dopo che il meccanismo avrà fatto esplorare tutti i nodi a sinistra della radice, la procedura che ha ricevuto *inizio* come puntatore e che è stata fino ad ora in attesa potrà continuare la sua esecuzione e chiamare sé stessa sul figlio di destra: di nuovo si mette in attesa per la fine dell'esecuzione di questa chiamata che prima farà esplorare tutti i nodi alla destra della radice. Solo allora la prima procedura terminerà 'raggiungendo il suo end'.

Quando la procedura è chiamata con NIL non fa niente ma si limita a terminare, senza richiamare più sé stessa ed interrompendo la catena delle chiamate. E' molto importante che si raggiunga una situazione di terminazione: diversamente si andrebbe avanti fino all'esaurimento della memoria messa a disposizione per la ricorsione (stack) mandando in crash il processo di esecuzione e, forse, causando anche il blocco dell'elaboratore.

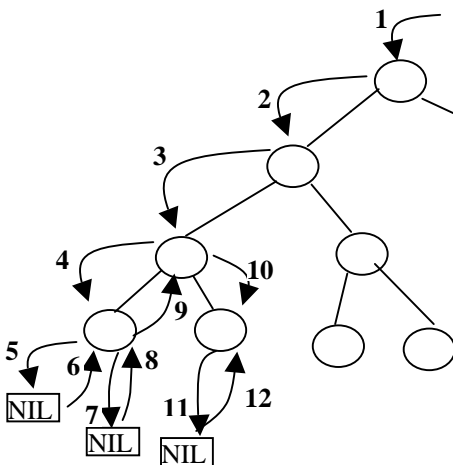
Un algoritmo ricorsivo corretto prima o poi invece si ferma perché raggiunge la condizione di terminazione che è chiamata la **base della ricorsione** (in realtà potrebbe ancora esaurire la memoria se ha bisogno di troppi passi per essere concluso, ma concettualmente l'algoritmo è corretto). Pensate alla base della ricorsione come al caso che termina la 'catena' delle chiamate. Nel nostro esempio la base è rappresentata da *nodo=NIL*: la condizione dell'*if* non è verificata e la procedura non fa nulla (soprattutto termina e non fa altre chiamate a se stessa).

E' anche importante notare che ogni chiamata ricorsiva impegna la procedura con un problema più semplice dell'originale: la prima procedura attivata deve stampare l'intero albero; la seconda il sottoalbero di sinistra, la terza il sottoalbero di sinistra ancora e così via fino a che, in profondità, una procedura riceverà un puntatore ad una foglia; a questo punto avviene l'ultima chiamata alla procedura che riceverà un puntatore NIL: la procedura non fa nulla e la 'catena' si ferma.

NOTA: LA PARTE CHE SEGUE (1 PAGINA) E' UN APPROFONDIMENTO PIUTTOSTO DIFFICILE DA SEGUIRE. E' FACOLTATIVA.

ATTENZIONE: DOPO LA PARTE FACOLTATIVA LA DISPENSA PROSEGUE!!

Esaminiamo in dettaglio cosa accade: la procedura viene invocata una prima volta e riceve il puntatore al primo nodo. Se questo è NIL (cioè se l'albero è vuoto) si ferma subito, come deve essere. Altrimenti stampa la parte informativa del primo nodo. Poi invoca se stessa sul figlio di sinistra con l'istruzione *StampaAlbero(nodo^.sx)*. A questo punto le procedure attivate sono due! La prima ad essere cronologicamente chiamata, *StampaAlbero(inizio)*, che **rimane in attesa** alla riga 2 aspettando il termine di *StampaAlbero(nodo^.sx)* che rappresenta la seconda procedura attivata in ordine di tempo. E' molto importante capire che l'istruzione della riga 3, *StampaAlbero(nodo^.dx)*, della prima procedura attivata non verrà eseguita fino a che non terminerà quella della riga 2.

La seconda procedura attivata inizia la sua esecuzione e stampa la parte informativa del nodo puntato dal parametro che ha ricevuto (se diverso da NIL). Poi effettua a sua volta una chiamata ricorsiva, attivando una terza copia della procedura *StampaAlbero* inviando come parametro il puntatore al suo figlio di sinistra. E così via. Ecco schematizzata la sequenza sul disegno dell'albero:

I numeri indicano ovviamente la sequenza cronologica. Solo quelli vicino alle frecce continue indicano però l'attivazione di una nuova 'incarnazione' della procedura. Le frecce tratteggiate indicano il raggiungimento della *base della ricorsione* che fa terminare, relativamente a quella parte dell'albero, la catena delle chiamate. Ad esempio, la procedura attiva al punto 4, dopo aver stampato la sua parte informativa fa la chiamata ricorsiva passando come parametro NIL (infatti non ha figli). La procedura chiamata (punto 5) non fa allora nulla e termina. Il controllo ritorna allora alla procedura del punto 4 (seguite la freccia tratteggiata numerata 6) che è in attesa alla riga 2 del codice e che può finalmente fare la seconda chiamata ricorsiva (corrispondente alla riga 3 del codice) passando ancora NIL (anche il figlio di destra non esiste). La procedura invocata (numerata con 7) termina e restituisce il controllo ancora una volta a quella numerata con 4. Quest'ultima ha infine esaurito le sue istruzioni e termina, restituendo il controllo a quella numerata con 3, che chiama 10, che chiama 11 (con nil) e così via …

Gestione dello stack per la ricorsione

Tutte queste 'incarnazioni' della stessa procedura condividono lo stesso codice. Non si pensi cioè che se il meccanismo ricorsivo ha bisogno di 20 chiamate, in memoria sia presente 20 volte il codice della procedura. Quello che accade è che quando un'incarnazione di una procedura ad un qualche passo della ricorsione ha bisogno di chiamare sé stessa, il suo *stato* (una sua copia delle variabili locali con i valori fino a quel momento calcolati e un 'segnalibro' all'istruzione cui era arrivata) è memorizzato in un area di memoria speciale, chiamata ***stack***. Quando il controllo è restituito, anche molti passi dopo, ad una procedura, il suo stato è ripreso dallo *stack* ed essa può continuare dal punto in cui era stata interrotta. Lo *stack* è una struttura dinamica ***LIFO*** (Last In First Out) in cui l'ultimo dato inserito è il primo che può essere ripreso. Per fare un paragone pensate ad una pila di piatti: l'ultimo piatto lavato sarà il primo ad essere ripreso per il risciacquo. Questo modo di funzionare è essenziale per poter accedere nell'ordine giusto agli stati delle diverse procedure: infatti è quella chiamata per ultima che riavrà il controllo per prima e continuerò dal punto in cui era stata interrotta. Osservando il grafico precedente: la 3 chiama la 4 (e lo stato della 3 viene salvato); la 4 chiama la 5 (e lo stato della 4 viene salvato); la 5 finisce: deve riprendere la 4, è il suo stato che serve, non quello della 3! La 4, chiamata per ultima, ha bisogno prima della 3 del suo stato: gli stati devono essere ripristinati nell'ordine inverso in cui sono stati salvati. Uno stack (realizzato come un vettore di byte in una zona della RAM) si comporta esattamente in questo modo. Nella simulazione sottostante, proc sta per procedura.

Proc 1 chiama proc 2

Lo stato di proc 1 viene salvato sullo stack

Proc 1 è messa in attesa

Stato proc 1

Proc 2 chiama proc 3

Lo stato di proc 2 viene salvato sullo stack

Proc 2 è messa in attesa

Proc 1 è sempre in attesa

Stato proc 2
Stato proc 1

Proc 3 chiama proc 4

Lo stato di proc 3 viene salvato sullo stack

Proc 3 è messa in attesa

Proc 1 e 2 sono sempre in attesa

Stato proc 3
Stato proc 2
Stato proc 1

Proc 4 chiama proc 5

Lo stato di proc 4 viene salvato sullo stack

Proc 4 è messa in attesa

Proc 1, 2 e 3 sono sempre in attesa

Stato proc 4
Stato proc 3
Stato proc 2
Stato proc 1

Proc 5 termina

Lo stato di proc 4 viene tolto dallo stack

Proc 4 riprende dalla riga alla quale si era interrotta

Proc 1, 2 e 3 sono sempre in attesa

Stato proc 3
Stato proc 2
Stato proc 1

Au… muso (email: …@….it sito web: www…)

RIPRENDE LA PARTE NON FACOLTATIVA DELLA DISPENSA!!

In alcuni problemi è proponibile sia una soluzione iterativa (con i cicli) che ricorsiva. Quest'ultima risulterà più lenta nell'esecuzione e consumerà più memoria (salvare lo stato occupa tempo e spazio) ma più naturale se il problema ha una natura essa stessa ricorsiva. In questi ultimi casi spesso la soluzione ricorsiva è MOLTO più semplice da programmare, al punto da risultare l'unica strada percorribile… La ricorsione si è rivelata uno strumento assai utile nel campo dell'Intelligenza Artificiale (IA) dove strutture complesse come gli alberi sono all'ordine del giorno per rappresentare la conoscenza ed il percorso decisionale per la soluzione di un problema.

QUALCHE ESEMPIO DI SOTTOPROGRAMMA RICORSIVO

Per definire un algoritmo ricorsivo dobbiamo sempre individuare:

La base della ricorsione: è il caso più semplice, quello per il quale sappiamo subito 'calcolare' il risultato. Quello a cui il meccanismo ricorsivo tenta di ricondursi un poco alla volta, passo dopo passo.

Il passo ricorsivo: quando non siamo di fronte al caso più semplice dobbiamo tentare di esprimerlo attraverso una 'formula' che richiama lo stesso sottoprogramma che stiamo scrivendo *ma con argomenti semplificati* che ci avvicinano al caso che rappresenta la base della ricorsione

Esempio 1: calcolo del fattoriale.

Indichiamo con *Fatt(N)* il fattoriale di un numero intero >=0. E' il prodotto di tutti gli interi da 1 a N. Ad esempio Fatt(6) = 6 * 5 * 4 * 3 * 2 * 1. Per definizione si stabilisce che Fatt(0) è 1. E' una funzione assai usata in matematica: ad esempio nel calcolo delle probabilità (combinazioni, disposizioni semplici o con ripetizione, coefficienti binomiali ecc.).

L'obiettivo è darne una definizione ricorsiva (esprimere un fattoriale usando ancora un fattoriale ma calcolato su un valore più semplice). Si ragiona così: qual è il caso più semplice di calcolo del fattoriale? Quando N è 0: infatti possiamo subito affermare senza bisogno di fare calcoli che il risultato è 1. E se dobbiamo calcolare Fatt(4)? Il risultato immediato non lo possiamo calcolare ma possiamo dire che è 4 * Fatt(3). Infatti Fatt(3)=3*2*1 e quindi 4*Fatt(3) equivale a 4 * 3*2*1, il calcolo di partenza.

Abbiamo allora ricondotto un caso più complesso, Fatt(4), al calcolo di un caso più semplice, Fatt(3). Come dire: non so calcolare direttamente Fatt(4) ma se riesco a calcolare Fatt(3) risalgo attraverso una moltiplicazione a Fatt(4). Fatt(3) è ancora troppo 'difficile' da calcolare ma lo si riconduce a Fatt(2): Fatt(3) = 3 * Fatt(2). Fatt(2) = 2 * Fatt(1). Fatt(1) = 1 * Fatt(0). Ma Fatt(0) so che vale uno senza bisogno di ricalcolare nessun altro fattoriale: ho raggiunto il caso base, la base della ricorsione che termina la catena delle chiamate ricorsive!

Schematicamente

$$Fatt(N) = \begin{cases} 1 \text{ se } N = 0 \\ N * Fatt(N-1) \text{ se } N>0 \end{cases}$$

```
function Fatt(N: integer): real;
begin
  if N=0 then
    Fatt:=1
  else
    Fatt:= N * Fatt(N-1)
end;
```

NOTA: come valore restituito ho scelto un *real* in quanto il calcolo del fattoriale molto rapidamente porta al superamento della capacità di una variabile *integer*.

 Autore: Fabrizio Camuso (email: camuso@camuso.it sito web: www.camuso.it)

Esempio 2: calcolo di x^y con x ed y interi positivi.

Indichiamo con *XallaY(x,y)* il sottoprogramma.

L'obiettivo è darne una definizione ricorsiva (esprimere una potenza usando ancora una potenza ma calcolata su un valore più semplice). Si ragiona così: qual è il caso più semplice ? Quando si chiede di elevare un numero alla zero. In questo caso infatti per definizione il risultato è 1. Un numero elevato alla prima può essere calcolato come il numero stesso moltiplicato la sua potenza zero. Infatti $x^1 = x * x^0 = x * 1 = x$. Abbiamo allora ricondotto un caso più complesso al calcolo di uno di cui sappiamo subito il risultato (x^0). Ora che sappiamo calcolare un numero elevato alla prima siamo in grado di calcolare i quadrati, Infatti un numero al quadrato può essere calcolato come il numero stesso moltiplicato per la sua potenza prima: $x^2 = x * x^1$. E così via: $x^3 = x * x^2 = x * x * x^1 = x * x * x * x^0$; $x^4 = x * x^3$ ecc.

Abbiamo allora ricondotto in generale un caso più complesso, x^y, al calcolo di un caso più semplice, $x * x^{y-1}$.
Schematicamente

$$XallaY(x,y) = \begin{cases} 1 \text{ se } y=0 \\ x * XallaY(x,y-1) \text{ se } y>0 \end{cases}$$

```
function XallaY(x,y: integer): real;
begin
  if y=0 then
    XallaY:=1
  else
    XallaY:= x * XallaY(x,y-1)
end;
```

NOTA: come valore restituito ho scelto un *real* in quanto il calcolo di una potenza molto rapidamente può portare al superamento della capacità di una variabile *integer*.

Esempio 3: calcolo di x*y con x ed y interi, y<>0.

Indichiamo con *XperY(x,y)* il sottoprogramma.

L'obiettivo è darne una definizione ricorsiva (esprimere un prodotto usando ancora un prodotto ma calcolato su un valore più semplice). Si ragiona così: qual è il caso più semplice ? Quando si chiede di moltiplicare un numero per 1: il risultato è il numero stesso. Se invece dobbiamo calcolare un numero per 2 possiamo esprimere il calcolo come il numero sommato al prodotto del numero per 1: x * 2 = x + (x*1). Di nuovo abbiamo ricondotto un prodotto complesso al calcolo di un prodotto più semplice di cui sappiamo subito il risultato (x*1). E così via: x * 3 = x + x*2 = x + x + x*1; x * 4 = x + x * 3 ecc.

Schematicamente

$$XperY(x,y) = \begin{cases} x \text{ se } y=1 \\ x + XperY(x,y-1) \text{ se } y>0 \end{cases}$$

```
function XperY(x,y: integer): real;
begin
  if y=1 then
    XperY:=x
  else
    XperY:= x + XperY(x,y-1)
end;
```

NOTA: come valore restituito ho scelto un *real* in quanto il calcolo di prodotto può portare abbastanza facilmente al superamento della capacità di una variabile *integer*.

Nota: matematicamente un prodotto è in effetti una somma ripetuta … x * 4 = x + x + x + x che è esattamente quanto fa la ricorsione!

Esempio 4: calcolo di x+y con x ed y interi.

Indichiamo con *XpiuY(x,y)* il sottoprogramma.

L'obiettivo è darne una definizione ricorsiva (esprimere una somma usando ancora una somma ma calcolata su un valore più semplice). Si ragiona così: qual è il caso più semplice ? Quando si chiede di sommare zero ad un numero: il risultato è il numero stesso. Se invece dobbiamo sommare 1 possiamo esprimere il calcolo come 1 più il risultato della somma tra il numero e 0; infatti: x + 1 = 1 + (x+0). Di nuovo abbiamo ricondotto un prodotto complesso al calcolo di un prodotto più semplice di cui sappiamo subito il risultato (x+0). E così via: x + 4 = 1 + (x+3) = 1 + 1 + (x+2) = 1 + 1 + 1 + (x+1) = 1 + 1 + 1 + 1 + (x+0) = 1 + 1 + 1 + 1 + x.

Schematicamente

$$XpiuY(x,y) = \begin{cases} x \text{ se } y = 0 \\ 1 + XpiuY(x,y-1) \text{ se } y>0 \end{cases}$$

```
procedure XperY(x,y: integer): integer;
begin
  if y=0 then
    XpiuY:=x
  else
    XpiuY:= 1 + XpiuY(x,y-1)
end;
```

Esempio 5: calcolo della serie di Fibonacci (esempio di doppia chiamata ricorsiva)

La serie di Fibonacci è la seguente: 0 1 1 2 3 5 8 13 21 34 55 … dove ogni elemento è la somma dei due precedenti (ad eccezione del primo e del secondo che sono per definizione 0 ed 1).

Indichiamo con *Fib(N)* l'ennesimo numero della sequenza (si parte dalla posizione 0!). Ad esempio Fib(0)=0; Fib(1)=1; 2; Fib(7)=13.

L'obiettivo è darne una definizione ricorsiva: un numero verrà appunto espresso come somma dei 2 numeri di Fibonacci che lo precedono, fino ai primi due che sono fissati per definizione. Fib(4) = Fib(3) + Fib(2) = Fib(2)+Fib(1) + Fib(1)+Fib(0) = Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0) = 3.

Schematicamente

$$Fib(N) = \begin{cases} 0 \text{ se } N = 0, 1 \text{ se } N=1 \\ Fib(N-2)+Fib(N-1) \text{ se } N>1 \end{cases}$$

```
procedure Fib(N: integer): integer;
begin
  if N=0 then
    Fib:=0
  else
    if N=1 then
      Fib:=1
     else
       Fib:=Fib(N-2) + Fib(N-1)
end;
```

Esercizi suggeriti:

- calcolo di A-B
- calcolo di A / B
- inversione di una stringa

LIMITI DELLA MEMORIA ALLOCATA STATICAMENTE

LIMITE 1

Consideriamo la seguente dichiarazione:

```
var
  voti: array[1..10] of integer;
```

Il programmatore sta indicando al compilatore che intende usare un vettore di 10 interi. Il compilatore di conseguenza, nella traduzione in linguaggio macchina, predispone (alloca) la quantità di RAM (byte) necessaria.

Con la maggior parte dei linguaggi di programmazione, quando il programma è mandato in esecuzione non c'è modo di mutare questa situazione (aumentare ad esempio il numero di elementi nel vettore). L'unico modo per aggiungere elementi al vettore è quello di modificare il sorgente e ricompilare il programma.

Immaginate la scena: un programmatore 'tirchio' dimensiona un vettore con troppo pochi elementi; un cliente, a mille chilometri di distanza, usa il programma in modo da esaurire il vettore: è costretto ad attendere l'intervento del programmatore, con grave disagio (ed arrabbiatura).

Esagerare con il numero di elementi del vettore potrebbe comportare uno spreco di RAM inaccettabile (se mediamente il numero di elementi usati del vettore è sensibilmente inferiore al massimo). Decidere invece di memorizzare i dati su disco farebbe crollare le prestazioni.

Naturalmente non saremo sempre così 'sfortunati': i mesi di un anno sono sempre dodici, ed i giorni al massimo 366; gli alunni di una classe difficilmente supereranno i quaranta, ecc. Diciamo che il problema è particolarmente evidente in quelle situazioni in cui il programmatore non può fare una stima esatta o quasi delle necessità.

Pensate, ad esempio, al gioco degli scacchi: è impossibile prevedere quante 'mosse' durerà la partita. Quanti elementi dovrà allora avere il vettore che memorizza le mosse? Se poi volessimo impostare un livello di gioco difficile il computer dovrà allora valutare molte posizioni per ogni mossa. Ma qualcuno preferirà un livello di gioco per principianti. Alcuni software permettono addirittura di giocare più partite in contemporanea. Quanta memoria RAM serve, quind,i per far funzionare il programma? Che cosa dovrebbe dichiarare il programmatore nella sezione VAR ?

> Lo spazio allocato staticamente in RAM non può essere modificato (aumentato o diminuito) durante l'esecuzione del programma.

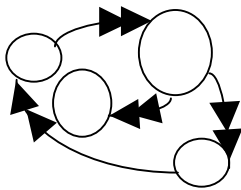
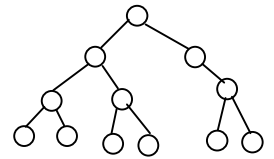
LIMITE 2

Immaginate ora di avere un elenco di nomi in un vettore e di averlo ordinato alfabeticamente (operazione 'costosa'). Immaginiamo anche che il vettore abbia dei 'posti liberi' in fondo. Ogni ulteriore inserimento rappresenta un'operazione piuttosto onerosa se vogliamo mantenere ordinato l'elenco. Infatti non basta aggiungere in fondo il nuovo nome: non è detto, infatti, che l'elenco rimanga ordinato. E' assai più probabile che il giusto posto per il nuovo nome sia in una posizione intermedia. Sarà allora necessario spostare verso destra di una posizione una parte degli elementi per fare spazio al nuovo. Come si diceva, l'inserimento in queste condizioni è un'operazione piuttosto onerosa ed accettabile solo se gli inserimenti a vettore già ordinato sono rari: purtroppo sono molteplici le applicazioni in cui è vero io il contrario ... Non va meglio nel caso si debba eliminare un elemento intermedio (perché?).

> Una struttura allocata staticamente è inefficiente per frequenti operazioni di inserimento ed eliminazione.

Autore: Fabrizio Camuso (email: camuso@camuso.it sito web: www.camuso.it)

LIMITE 3

Date un’occhiata alla struttura dati disegnata qui a lato (si chiama albero). Essa è adatta per rappresentare situazioni in cui esiste una gerarchia tra gli elementi da memorizzare (struttura directory, organigramma aziendale, mosse e contromosse in un gioco ecc.). Beh, penso sarete d’accordo nell’affermare che non è facile rappresentarla usando gli array. E che dire in merito alla rappresentazione di una rete stradale: in questo caso non c’è addirittura nessun ordine tra i nodi da collegare!

Una struttura allocata staticamente è inadeguata per rappresentare strutture non lineari.

ALLOCAZIONE DINAMICA DELLA MEMORIA

Il programmatore può invece scegliere, nei casi in cui risulta conveniente, di gestire la memoria dinamicamente. Significa che avrà a disposizione delle istruzioni con cui comandare durante l’esecuzione del programma la allocazione di altro spazio per le variabili ma anche per restituire lo spazio che non serve più.

UN’ APPLICAZIONE CONCRETA

Immaginiamo che ci sia stato commissionato un programma per monitorare in tempo reale gli ingressi allo SMAU. In particolare si vogliono inserire le località di provenienza dei visitatori in un elenco che deve essere consultabile con la massima velocità per poter effettuare statistiche in tempo reale. Da subito viene scartata l’idea di utilizzare i file: troppo lenti. Si utilizzerà la RAM cercando comunque di non ‘sprecarla’ per non compromettere le prestazioni del sistema. Si decide di organizzare le informazioni in questo modo: blocchi (vettori) da 20 stringhe ciascuno fino ad un massimo di 1000 blocchi. Fissiamo anche la lunghezza di ogni stringa: 50 caratteri. All’inizio si comincia con un solo blocco (vettore): esaurito il primo si chiederà memoria per il secondo e così via ...

Rifletti ...
Capita la differenza? Non verranno predisposti da subito 1000 vettori da 20 stringhe! Verrà dato spazio ai vettori in RAM progressivamente, man mano che se ne avvertirà l’esigenza ...

L’uso della memoria dinamica consentirà di occupare all’inizio solo la memoria necessaria per le 20 stringhe del primo blocco e di occuparne dell’altra solo quando veramente necessario. E’ facile convincersi che in questo modo la memoria eventualmente sprecata è quella corrispondente a 19 stringhe, nell’ipotesi di usare solo la prima stringa dell’ultimo blocco allocato.

Rifletti ...
Pur non essendo ottimale (rimane il limite delle 20*1000 blocchi=20000 stringhe) in questo modo riusciamo a gestire una GROSSA quantità di stringhe ottimizzando l’uso della memoria. In seguito estenderemo la tecnica per gestire elenchi ‘infinitamente’ lunghi (ad esaurimento RAM ... o spazio su disco visto che tutti i moderni sistemi operativi supportano la gestione virtuale della RAM usando il disco come estensione ‘lenta’ di quest’ultima)

 Autore: Fabrizio Camuso (email: camuso@camuso.it sito web: www.camuso.it)

Procediamo con gradualità. Prima preoccupiamoci della gestione dinamica di un singolo vettore di 20 stringhe, cioè di un solo blocco. Poi estenderemo la gestione a 1000 blocchi. Come si fa a chiedere al Pascal di allocare spazio per un vettore di 20 stringhe? Ed il vettore si userà come tutti gli altri vettori? Un passo alla volta ...

Iniziamo con il definire un tipo che rappresenta un singolo blocco:

```
type
  vettore = array[1..20] of string[50];  (* il punto di partenza è un normale vettore di stringhe *)

  pun = ^vettore;         (* il tipo blocco  'punta' ad oggetti di tipo vettore; si parla di tipo puntatore *)
                          (* il carattere ^ è fondamentale per indicare che si tratta di un puntatore! *)
var
  blocco: pun (* variabile puntatore che memorizzerà l'indirizzo di un blocco quando sarà creato *)
```

La variabile blocco è di tipo pun, cioè (risalendo la catena delle dichiarazioni) è un puntatore ad oggetti di tipo vettore (si scrive ^vettore). Al momento non sta puntando a nulla perché il programmatore non ha ancora chiesto l'allocazione di memoria per il blocco di stringhe.

NOTA BENE: blocco non può essere usata come una variabile qualsiasi. Ad esempio producono un errore le seguenti istruzioni:

NO !!!   writeln(blocco) o readln(blocco) o blocco:=13;   NO !!!

Un puntatore è una variabile speciale. Serve a contenere indirizzi di oggetti creati nella RAM. Quindi prima deve essere creato un oggetto e poi si memorizza nel puntatore dove inizia in memoria quell'oggetto. Usando poi il puntatore potremo scrivere dati in quel blocco di memoria, recuperarli, distruggere il blocco di memoria quando non servirà più.

APPROFONDIMENTO

La dimensione effettiva di un puntatore dipende da come il processore gestisce gli indirizzi di memoria e dalla capacità del compilatore di sfruttare appieno le possibilità offerte da un processore. Il Turbo Pascal è in grado di gestire sia indirizzi di 2 byte, cioè 16 bit (memoria di al massimo 2^16 – 1 = 64Kbyte, sia indirizzi di 4 byte (32bit, memoria segmentata DOS) indicandoli con una coppia numero di segmento (64K possibili segmenti) + posizione nel segmento (da 0 a 64K).

Per chiarire in modo inequivocabile che un puntatore non sta individuando nessun oggetto in memoria lo si inizializza con un valore speciale: NIL (nulla).

```
blocco := NIL;
```

In questo modo il programmatore può verificare se il puntatore sta puntando ad un oggetto valido (uno degli errori più pericolosi che si possano commettere è infatti l'uso di un puntatore che non sta puntando a nulla):

```
if blocco<>NIL then ... uso 'sicuro' del puntatore ...
```


Autore: Fabrizio Camuso (email: camuso@camuso.it sito web: www.camuso.it)

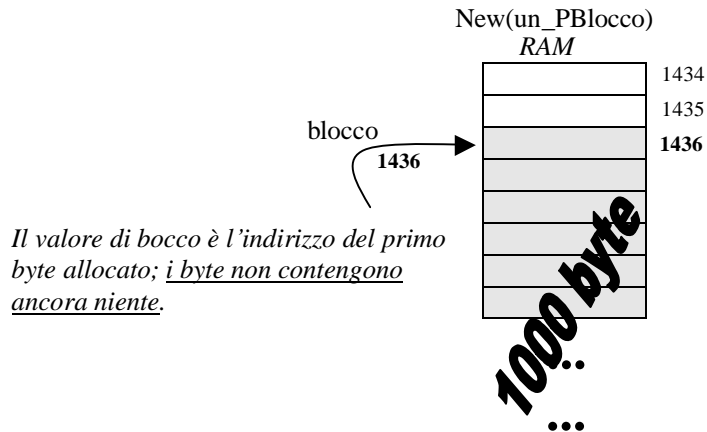
L'istruzione new: allocazione della memoria

Torniamo all'esempio precedente e continuiamo dal punto in cui ci eravamo interrotti

```
type
   vettore = array[1..20] of string[50];
   pun = ^vettore;

var
  blocco: pun
```

E' arrivato il momento di creare un blocco (vettore) e di memorizzare il suo indirizzo nella variabile blocco.

```
   new(blocco);
```

Letteralmente: crea un nuovo oggetto del tipo individuato dalla variabile puntatore. Poiché blocco è un puntatore ad oggetti di tipo vettore e questi ultimi sono array di20 stringhe, è appunto un vettore di 20 stringhe che viene creato in memoria. Poiché ogni stringa occupa 50 carratteri, il blocco complessivo di byte occupati è 20*50=1000 byte.

Ecco quello che accade più in dettaglio: il sistema operativo, se la memoria non è esaurita, riserva un blocco di tanti byte consecutivi quanti sono quelle necessari a memorizzare 20 stringhe da 50 caratteri (circa 1000 byte). Non è dato sapere a priori dove si troverà nella RAM questo blocco ma ciò che conta è sapere l'indirizzo del suo primo byte: è il suo indirizzo ad essere memorizzato nella variabile puntatore blocco. Considerando la situazione rappresentata nel disegno soprastante, ho immaginato che il sistema operativo abbia trovato posto a partire dal byte all'indirizzo 1436: è questo il valore che viene memorizzato nella variabile blocco. Se non c'è spazio il sistema operativo restituisce NIL.

E' molto importante capire che al momento della scrittura del programma il vettore non esiste ancora. E neppure alla partenza del programma: è necessario il comando esplicito *new*. NOTA: la new deve essere comandata una sola volta per tutto il vettore. Nel caso il vettore non servisse più sarà possibile restituire la memoria al sistema operativo e riutilizzare il puntatore blocco per un altro vettore (subito o in un secondo momento).

Utilizzo della memoria allocata

Immaginando di aver già creato l'oggetto vettore, proviamo a memorizzare una stringa nel suo primo elemento:

```
  blocco^[1]:='Sono la prima stringa del blocco';
```


Autore: Fabrizio Camuso (email: camuso@camuso.it sito web: www.camuso.it)

Attenzione a non usare blocco come se fosse esso stesso il vettore di stringhe:

blocco[1]:='una stringa' NO!!

Il puntatore infatti non è l'oggetto ma l'indirizzo dell'oggetto. Per passare dall'indirizzo all'oggetto è necessario indicare il simbolo ^ dopo il nome del puntatore. In progressione: blocco è l'indirizzo, blocco^ è l'oggetto puntato, cioè il vettore e blocco^[1] è il suo primo elemento

Ed ora un ciclo for per valorizzare a stringa vuota i restanti elementi

```
for i:=2 to 20 do
  blocco^[i] := '';
```

Ed un altro ciclo for che visualizza tutte le stringhe:

```
for i:=1 to 20 do
  writeln( blocco^[i] );
```

L'istruzione dispose: restituzione della memoria

Quando il vettore ha esaurito la sua utilità possiamo restituire con il comando dispose la memoria al sistema operativo, cioè deallocarla:

```
 dispose(blocco); blocco:=nil;
end.
```

NOTA BENE: dopo la dispose il puntatore perde di significato. E' un GRAVE errore tentare di usare un puntatore deallocato. Purtroppo, il Turbo Pascal non mette automaticamente a nil un puntatore deallocato (con altri linguaggi accade) ed è allora buona pratica farlo personalmente.

E' altrettanto GRAVE deallocare un puntatore che non punta a niente (NIL): di solito va in crash il programma.

Rifletti ...

Il fatto che la memoria possa essere restituita quando non serve più rappresenta già un grosso vantaggio rispetto all'uso di un normale vettore di stringhe. Il prezzo da pagare è una certa complicazione delle operazioni e, cosa ben più tangibile, un maggiore rischio di commettere errori gravi.

In effetti se la quantità di memoria non rappresenta un problema conviene sovradimensionare gli array standard ed evitare complicazioni. D'altra parte in molte situazioni la quantità di memoria è il problema principale (provate a pensare se un programma di grafica tipo PhotoShop dovesse allocare decine di megabyte di RAM per niente per ogni immagine ad alta risoluzione che poi non sarà veramente caricata in memoria...).

ATTENZIONE: SE AVETE CAPITO GLI ARGOMENTI FIN QUI ESPOSTI AVETE RAGGIUNTO COMPLETAMENTE GLI OBIETTIVI!!

IL SEGUITO PUO' ESSERE CONSIDERATO UN APPROFONDIMENTO, NON ALLA PORTATA DI TUTTI.

approfondimento

Dopo le prime 20 stringhe avremmo esaurito lo spazio, nè più nè meno che con un vettore classico. La soluzione consiste nell'usare un vettore di puntatori, di 1000 puntatori per l'esattezza.

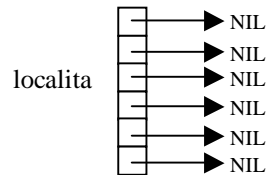
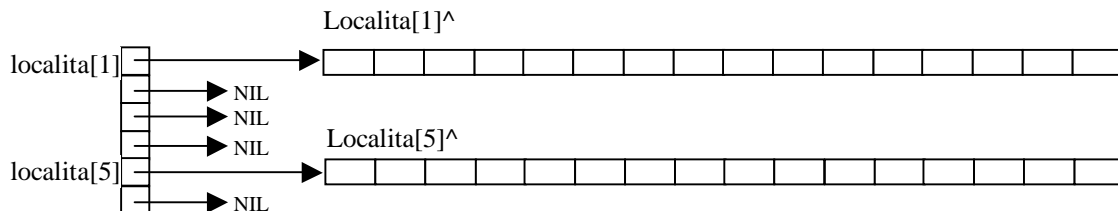
Intanto è necessario dichiarare il vettore di puntatori:

```
type
  P1000Blocchi=array[1..1000] of blocco;

var
    Localita: P1000Blocchi;
```

Ora abbiamo a disposizione il tipo vettore di 1000 puntatori a blocchi di 20 stringhe ciascuno e relativa variabile.

```
 (* inizializzo il vettore di puntatori *)
  for i:=1 to 1000 do
    Localita[i]:=nil;
```

NIL

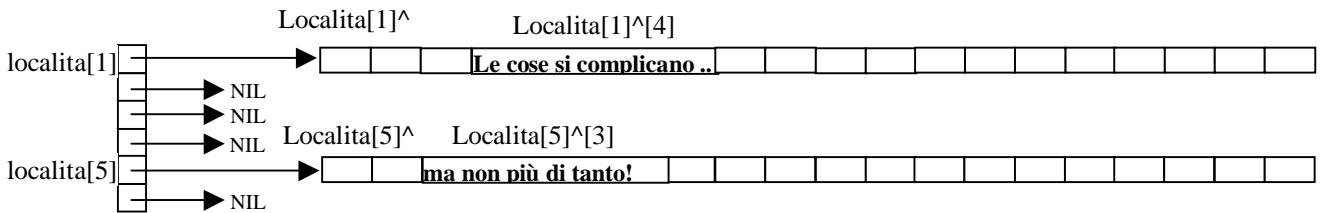
E' utile rappresentare graficamente la situazione (solo con i primi elementi del vettore di puntatori):

```
(* chiediamo spazio per due vettori: il primo ed il quinto, senza un motivo particolare, giusto per prova *)
  new( Localita[1] );
  new( Localita[5] );
```

Solo due puntatori hanno a questo punto collegati due blocchi. Tutte le caselle dei vettori contengono valori casuali.

```
(* inizializzo le stringhe dei vettori, ma solo per i puntatori a blocco diversi da nil, ovviamente *)
  for i:=1 to 1000 do
    if Localita[i]<>nil then
      for j:=1 to 20 do
        Localita[i]^[j]:='';
```

Ora tutte le caselle dei vettori contengono la stringa nulla. Notate il controllo if localita[i]<>NIL che seleziona solo gli elementi del vettore di puntatori che hanno collegato veramente un vettore di stringhe (il primo ed il quinto elemento, nel nostro caso).

approfondimento
approfondimento

Localita è il nome di un vettore. Localita[i] accede all'i-mo elemento di quel vettore (un puntatore). Localita[i]^ accede all'elemento puntato che è a sua volta un vettore di stringhe. Localita[i]^ [j] è il j-mo elemento di questo vettore, cioè la j-ma delle 20 stringhe di quel vettore. Vi gira la testa?

```
 (* memorizzo una stringa nel quarto elemento del primo vettore *)
 Localita[1]^[4]:='le cose si complicano ...';

 (* ed una nel terzo elemento del quinto vettore *)
 Localita[5]^[3]:='ma non piu'' di tanto!';
```

```
 writeln('--------------------------');   writeln('Stringhe nel vettore n. 1: ');   writeln('--------------------------');

 for i:=1 to 20 do
   writeln( (Localita[1]^)[i] );
 readln;

 writeln('--------------------------'); writeln('Stringhe nel vettore n. 5: '); writeln('--------------------------');

 for i:=1 to 20 do
   writeln( Localita[5]^[i] );

 readln;

(* deallochiamo i vettori di stringhe *)
for i:=1 to 1000 do
 if localita[i]<>nil then
 begin
   dispose(localita[i]);
   localita[i]:=nil
 end
end.
```

Notate , di nuovo, il controllo sul puntatore a NIL prima di deallocare. Deallocare con dispose un puntatore a NIL è un GRAVE errore e di solito porta al crash del programma.

Rifletti ...

Quanta memoria può far risparmiare la gestione dinamica? A volte moltissima ! Se ad esempio si scoprisse che mediamente solo metà delle stringhe sono necessarie, la gestione statica staticamente userebbe sempre e comunque 1.000.000 di byte (20.000 stringhe x 50 caratteri) mentre quella dinamica circa la metà: 520.000 (2 byte a puntatore x 1000 puntatori + 10.000 stringhe x 50 caratteri) !!

Risparmi a parte la gestione dinamica restituisce memoria nei momenti in cui non serve e migliora le prestazioni del sistema operativo e, quindi, dell'elaboratore.

QUI TERMINA L'APPROFONDIMENTO.

Gli errori più comuni

- Usare un puntatore prima di averlo ‘agganciato’ con l’istruzione new ad un blocco di memoria valido

- Usare un puntatore dimenticandosi il simbolo di puntatore (^):

  se p è un puntatore ad un intero:

  new(p);
  SI’: p:=3;   NO: p^:=3

- Usare un puntatore dopo averlo ‘sganciato’ con l’istruzione dispose dal blocco di memoria cui puntava:

  se p è un puntatore ad un intero:

  new(p);
  p^:=3;
  dispose(p);   ora p non punta più a nulla e non è più utilizzabile

  writeln(p^);
  NO !! p non ha più significato: l’area di memoria cui puntava è stata restituita con dispose al sistema operativo che potrebbe utilizzarla per altri scopi; usare l’indirizzo non più valido contenuto in p potrebbe portare a conseguenze anche molto gravi per la stabilità del sistema.

- Sbagliare la posizione del simbolo di puntatore (^) con vettori o record. Ecco una rassegna di casi:

```
TYPE
  p= ^string;

  vett : array[1..10] of p;

rec=record
   codice: integer;
   pun: p
end;

vetRec=array[1..10] of rec;

recVet=record
  codice: integer;
  v: array[1..10] of p
end;

VAR
   vpun: vett;
   unRec: rec;
   vrec: vetRec;
   unRecVet: recVet;
```

SI': new(vpun[3]);   SI': vpun[3]^:='ciao'; NO: new(vpun)
NO: vpun^[3]:='ciao'; (giusto se vpun fosse un puntatore a vettore di stringhe e non un vettore di puntatori)

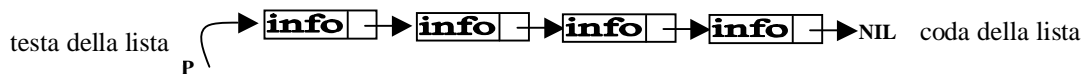
SI': new(unRec.pun); SI': unRec.pun^:='ciao';
NO: unRec^.pun:='ciao'; (giusto se unRec fosse un puntatore ad un record contenente una stringa pun)

SI': new(vrec[2].pun); SI': vrec[2].pun^:='ciao';
SI': new (unRecVet.v[2]);   SI': unRecVet.v[2]^:='ciao';

In generale è sufficiente ricordarsi di indicare il simbolo di puntatore (^) dopo il nome del puntatore (con la particolarità che v[i] e non 'v' è il nome del puntatore quando 'v' è un vettore.

LISTE SEMPLICI

Pur con i vantaggi visti, un limite alle stringhe gestibili nell'esempio precedente esiste comunque: ventimila. Vediamo di superarlo. Realizzeremo una concatenazione di elementi (lista) che potrà essere espansa senza limiti (RAM permettendo).

testa della lista P → info → info → info → info → NIL coda della lista

Nel disegno P rappresenta il puntatore che individua l'ingresso alla lista; ogni elemento (nodo) della lista ha una parte informatica (INFO) che può essere qualsiasi cosa: un semplice intero, una stringa, un vettore ecc. ogni nodo è collegato al successivo con un puntatore. Il punto di ingresso individua la testa della lista e all'altro capo troviamo invece la cosiddetta coda della lista.

Una lista ha anche altri vantaggi, oltre a quello di poter 'crescere' indefinitamente: se gli elementi sono in ordine (alfabetico, per esempio) l'aggiunta di un nuovo elemento, mantenendo l'ordinamento, è questione di pochissime operazioni. Anche l'eliminazione di un elemento ha un costo estremamente inferiore rispetto ai vettori.

<u>BASTA ARRAY?</u> Il fatto che la lista sia 'forte' proprio dove gli array sono 'deboli' non significa che debba essere d'ora in poi preferita ad essi (anzi!). Le liste hanno purtroppo delle controindicazioni, che sono comuni a tutte le strutture realizzate con la memoria dinamica: , ma è ancora troppo presto per parlarne…

Graficamente rappresenteremo le liste come segue:

P → NIL

Quando la lista è vuota. *P*, il puntatore che rappresenta il suo punto di ingresso, vale *NIL*. Si arriva in questa situazione o perché ancora nessun elemento è stato inserito alla lista o perché sono stati tutti eliminati.

Quando la lista è formata da un solo elemento, puntato da *P*. L'elemento è costituito logicamente da due parti: quella informativa ed un puntatore all'elemento che eventualmente segue nella lista (vale *NIL* se si è alla fine)

Una lista in uno stato intermedio.

La lista come esempio di ADT

La lista è un esempio di ADT (Abstract Data Type, Tipo di Dato Astratto): nuovi tipi definiti dal programmatore per i quali è definito un ben preciso insieme di operazioni.

Ai programmatori non interessa come sia realizzato internamente l'ADT (potrebbe essere costruito usando vettori piuttosto che record per memorizzare i dati, usare un metodo di ordinamento piuttosto che un altro ecc.). Al programmatore interessa solo sapere come ottenere i servizi di un ADT cioè quali comandi può inviare ad un esemplare di un ADT e quali risultati ottiene.

Per fare un esempio, è come se il Pascal offrisse solo il tipo integer, con le operazioni che tutti conoscete. Il programmatore potrebbe aggiungere l'ADT real o string con tutte le operazioni note. Chi userà gli ADT non si preoccuperà di come i real e le string sono memorizzate e gestite: gli basta sapere come creare variabili di quei tipi, come assegnare loro valori, fare operazioni con esse, come visualizzarle sullo schermo ecc. E non sarà possibile chiedere operazioni non previste per quell'ADT (moltiplicare due stringhe, per esempio).

Per l'ADT lista semplice potremmo definire almeno le seguenti operazioni: aggiunta di un elemento all'inizio o alla fine della lista (o in punto intermedio), aggiunta di un elemento mantenendo la lista in ordine, eliminazione di un elemento, estrazione (l'elemento tolto non viene distrutto ma resta a disposizione del programmatore), ricerca di un elemento, visita di tutti gli elementi di una lista, controllo per sapere se la lista è vuota, distruzione della lista

Prima di scrivere le operazioni vediamo come rappresentare gli elementi della lista: un elemento deve contenere due cose, la parte informativa ed il puntatore all'elemento che lo segue nella lista. Sono due informazioni completamente diverse, per cui la struttura dati più naturale da usare è il record.

Verrebbe spontaneo scrivere la dichiarazione del record nel modo seguente (immaginando che la parte informativa sia un semplice intero, realizzando così una lista di integer ...):

```
type
 elemento=record
   inf: integer;          (* informazione *)
   pun: ^elemento    (* puntatore *)
 end;
```

NO !!

Da un punto di vista logico è corretto: pun è il puntatore all'elemento successivo che è in effetti di tipo elemento, per cui un puntatore a questo tipo si scrive proprio ^elemento. Purtroppo il compilatore non conosce ancora esattamente cosa sia elemento quando dichiariamo pun perché stiamo proprio definendo elemento e non abbiamo ancora finito! Capite? Ci stiamo mordendo la coda: per definire elemento dovremmo già sapere cosa sia elemento!

Il Pascal offre una scappatoia: prima della definizione di elemento possiamo definire un tipo puntatore ad esso. Si chiama dichiarazione forward (in avanti). E' un'eccezione: il compilatore sa che dovrebbe trovare la dichiarazione di cosa sia elemento <u>dopo:</u>

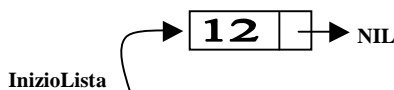
```
type
puntatore= ^elemento; (* OK: il compilatore si aspetta di trovare dopo cosa sia elemento *)

elemento=record
   inf: integer;       (* informazione *)
   pun: puntatore    (* il compilatore sa già cos'è il tipo puntatore *)
 end;
```

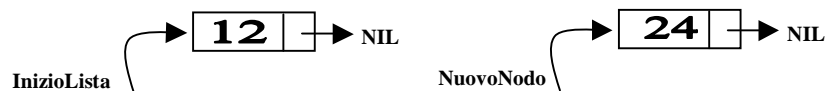
Una volta definiti i tipi è necessario dichiarare una variabile che rappresenta l'inizio della lista:

```
 var
   InizioLista: puntatore;
```

Proviamo a creare un nodo isolato di questa lista:

```
begin
  (*creiamo il primo nodo*)
   new(InizioLista);

   InizioLista^.inf:=12;
   InizioLista^.pun:=nil;
```

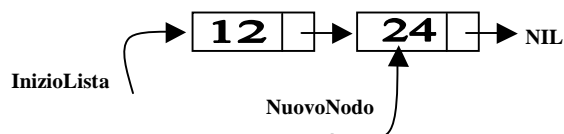
InizioLista è un puntatore. *InizioLista^* è l'oggetto puntato, un record: per accedere ai suoi campi informativi si usa la notazione 'punto'. *InizioLista^.inf* è quindi la sua parte informativa, *InizioLista^.pun* quella puntatore. Dopo queste istruzioni possiamo rappresentare graficamente la lista così:

Creiamo un secondo elemento (per il momento non collegato al primo):

```
var
   InizioLista: puntatore;
   NuovoNodo: puntatore:
       ...

  new(NuovoNodo);
  NuovoNodo^.inf:=24;
  NuovoNodo^.pun:=nil;
```

NOTA BENE Non è possibile usare ancora InizioLista per creare il secondo nodo: perderemmo il primo:

```
new(InizioLista);
InizioLista ^.inf:=24;
InizioLista ^.pun:=nil
```

Da qualche parte in memoria i dati del primo elemento ci sono ancora: avendo però cambiato l'indirizzo memorizzato nel puntatore con la seconda new, sono diventati irraggiungibili!! InizioLista porta ora al secondo nodo, perdendo la possibilità di usare il primo.

Ora concateniamo i due nodi, agganciando il secondo al primo:

```
InizioLista^.pun:=NuovoNodo;
```

Nella casella puntatore del primo nodo (la casella dopo il 12) viene memorizzato l'indirizzo del secondo (NuovoNodo)

In effetti il secondo nodo è raggiunto in questo momento da due puntatori: quello usato per crearlo e quello del nodo che lo precede (realizzando così la catena).

Dopo aver visto i meccanismi fondamentali, siete pronti per l'analisi di un programma abbastanza completo di gestione delle liste semplici. Naturalmente tutte le operazioni verranno implementate come sottoprogrammi! Infatti sarebbe molto sconveniente scrivere un codice apposito per aggiungere il terzo elemento e poi il quarto ecc.

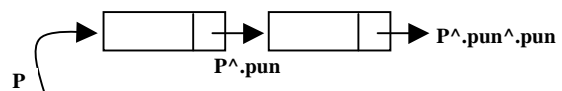
Per brevità non viene mai effettuato il controllo sul buon esito delle new.

```
Program liste_semplici;
uses crt;
type
  puntatore= ^elemento;
  elemento=record
    inf: integer;       (* informazione *)
    pun: puntatore    (* puntatore *)
end;
```

STAMPA DELLA LISTA (IN EFFETTI E' UNA VISITA COMPLETA DI TUTTI GLI ELEMENTI DI UNA LISTA)

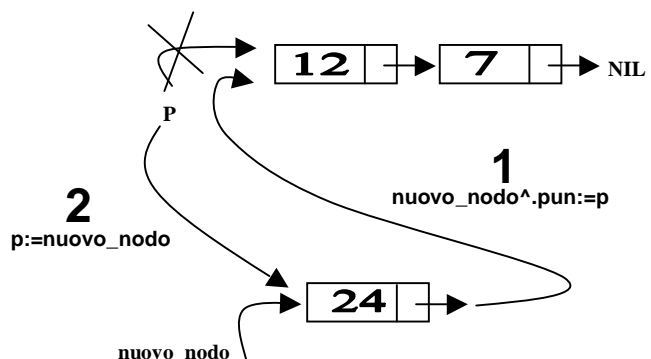
```
  procedure stampa_lista(p: puntatore);
  begin
    writeln('----');

    while p<>nil do
    begin
      write(p^.inf,'  ');
      p:=p^.pun
    end;

    writeln('----');
  end;
```

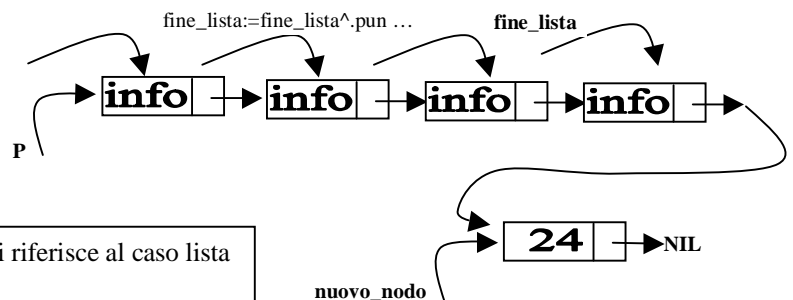
p è il puntatore all'inizio della lista;

p:=p^.pun è l'istruzione classica con cui si passa da un nodo al successivo. Se infatti *p* punta ad un certo nodo, la scrittura *p^.pun* corrisponde al contenuto della sua casella puntatore, cioè l'indirizzo del nodo successivo:

NB: *p* è passato per valore; possiamo quindi modificarlo senza timore di perdere l'inizio della lista…

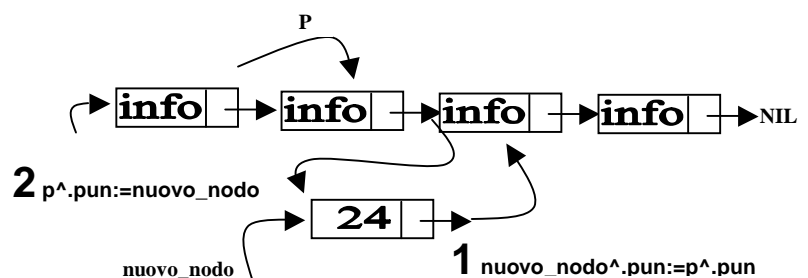
INSERIMENTO IN TESTA DI UN NUOVO ELEMENTO: riceve il puntatore 'p' ad una lista ed un valore da inserire

```
procedure inserisci_in_testa(var p:puntatore; valore:integer);
  var nuovo_nodo: puntatore;
  begin
    new(nuovo_nodo); nuovo_nodo^.inf:=valore;
    nuovo_nodo^.pun:=p;
    p:=nuovo_nodo;
 end;
```

E' importante prima salvare il valore di *p* nella casella puntatore del nuovo nodo: diversamente (invertendo cioè l'ordine dei due assegnamenti) si perderebbe il valore originale di *p* cioè del primo nodo, rendendo irraggiungibile l'intera lista!!

```
(*INSERIMENTO IN CODA DI UN NUOVO ELEMENTO: riceve il puntatore 'p' ad una lista ed un valore da inserire *)
 procedure inserisci_in_coda(var p:puntatore; valore:integer);
 var nuovo_nodo,fine_lista: puntatore;
 begin
   new(nuovo_nodo); nuovo_nodo^.inf:=valore;
   nuovo_nodo^.pun:=nil;

   (* devo distinguere il caso lista vuota *)
   if p<>nil then  (* lista non vuota *)
   begin
     fine_lista:=p; (* parto dall'inizio ... *)
     while fine_lista^.pun<>nil do    (* ... ed avanzo fino alla fine ... *)
        fine_lista:=fine_lista^.pun;

     fine_lista^.pun:=nuovo_nodo
   end
   else
     p:=nuovo_nodo
 end;
```

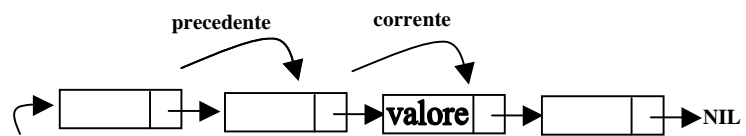
Anche se sarà poi necessario differenziare il caso lista vuota, queste istruzioni vanno bene in entrambi i casi.

Il disegno qui a lato si riferisce al caso lista non vuota…

INSERIMENTO DI UN NUOVO ELEMENTO IN UNA POSIZIONE INTERMEDIA: riceve il puntatore 'p' al nodo che precederà quello da inserire

```
 procedure inserisci_in_mezzo(var p:puntatore; valore:integer);
 var nuovo_nodo: puntatore;
 begin
   new(nuovo_nodo); nuovo_nodo^.inf:=valore;
   if p=nil then  (* la lista e' vuota *)
   begin
     nuovo_nodo^.pun:=nil;  p:=nuovo_nodo
   end
   else
   begin
     nuovo_nodo^.pun:=p^.pun;    (* aggancio successivo *)
     p^.pun:=nuovo_nodo          (* mi faccio puntare da precedente *)
   end
 end;
```

Il disegno qui a lato fa evidentemente riferimento al caso lista non vuota…

RICERCA IN UNA LISTA NON ORDINATA

La procedura non si limita a restituire un puntatore all'elemento eventualmente trovato ma anche il puntatore a quello che eventualmente lo precede. Ho scelto questo comportamento perché per alcune operazioni, una volta trovato un elemento, è necessario disporre anche del puntatore all'elemento precedente (ad esempio nel caso volessimo cancellare l'elemento trovato)

```
procedure cerca_non_ordinata(p: puntatore; valore: integer; var corrente,  precedente: puntatore);
 begin
   precedente:=nil; corrente:=nil;
   if p<>nil then
   begin
     corrente:=p;
     while (corrente^.inf<>valore) and (corrente^.pun<>nil) do
     begin
       precedente:=corrente;
       corrente:=corrente^.pun
    end;

    if corrente^.inf<>valore then
    begin
      precedente:=nil;
      corrente:=nil
    end
  end
 end;
```

La ricerca è effettuata su una lista non ordinata. Se *valore* viene trovato, in *corrente* viene restituito il puntatore al nodo cercato, in *precedente* quello al nodo che lo precedete (*nil* se il nodo cercato è il primo). Se il valore non viene trovato sia *corrente* che *precedente* sono messi a *nil*).

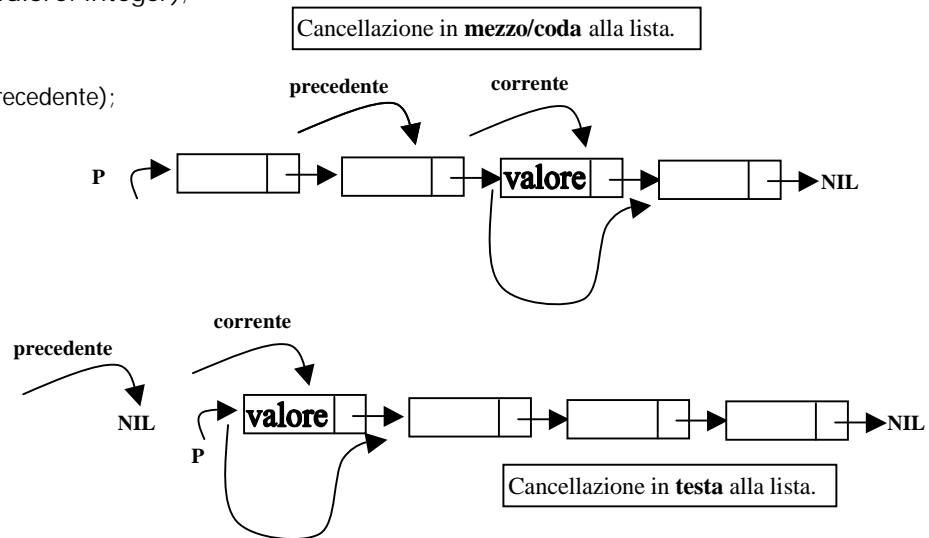
RICERCA IN UNA LISTA ORDINATA

Questa versione della ricerca lavora su una lista ordinata (diciamo in modo crescente). Trovato un elemento il cui valore supera quello che si sta cercando non ha più senso continuare la ricerca … quelli seguenti saranno per forza tutti maggiori!

```
procedure cerca_ordinata(p: puntatore;  valore: integer; var corrente, precedente: puntatore);
 begin precedente:=nil; corrente:=nil;
   if p<>nil then
   begin
     corrente:=p;
     while (corrente^.inf<valore) and (corrente^.pun<>nil) do
     begin
       precedente:=corrente;
       corrente:=corrente^.pun end;
     if corrente^.inf<>valore then
     begin precedente:=nil; corrente:=nil end end end;
```

ELIMINAZIONE DI UN ELEMENTO

Questa procedura invoca quella di ricerca per ottenere il puntatore al nodo da cancellare ed a quello che lo precede (se esiste). Sono infatti queste le informazioni necessarie per una corretta eliminazione.

```
procedure elimina(var p: puntatore; valore: integer);
var  corrente,precedente: puntatore;
begin
   cerca_non_ordinata(p,valore,corrente,precedente);
  if corrente<>nil then
  begin
    if precedente<>nil then
      (* eliminazione in mezzo o in coda *)
      precedente^.pun:=corrente^.pun
    else    (* eliminazione in testa *)
      p:=corrente^.pun;
    dispose(corrente)
  end
end;
```

INSERIMENTO IN UNA LISTA ORDINATA

Il nuovo elemento deve essere inserito mantenendo la lista ordinata.

```
procedure inserisci_in_ordine(var p:puntatore; valore:integer);
  var nuovo_nodo,precedente,corrente: puntatore;
  begin
    new(nuovo_nodo); nuovo_nodo^.inf:=valore;
    if p=nil then
    begin
      nuovo_nodo^.pun:=nil;  p:=nuovo_nodo
    end
    else
      if valore<p^.inf then  (* inserzione in testa *)
      begin
        nuovo_nodo^.pun:=p;  p:=nuovo_nodo
      end
      else
      begin
        (* cerca il punto di inserimento *)
        precedente:=p; corrente:=p;
        while (corrente^.inf<=valore) and (corrente^.pun<>nil) do
        begin
          precedente:=corrente;  corrente:=corrente^.pun
        end;
        if valore>corrente^.inf then  (* inserimento in coda *)
        begin
          nuovo_nodo^.pun:=nil; corrente^.pun:=nuovo_nodo
        end
        else (* inserimento in mezzo *)
        begin
          nuovo_nodo^.pun:=corrente;  precedente^.pun:=nuovo_nodo
        end  end   end;
```

Inserisce nuovi valori mantenendo un ordinamento crescente. La tecnica usata è la stessa del sort per inserzione (ricerca del giusto punto di inserimento) usata per i vettori ma grazie ai puntatori è evitato tutto il sovraccarico dello spostamento degli elementi per far posto a quello nuovo!

ESTRAZIONE DI UN ELEMENTO

Estrai si differenzia dalla elimina perché 'sgancia' il nodo che contiene l'informazione cercata ma non lo distrugge: restituisce il puntatore al nodo ed è poi responsabilità del chiamante liberare la memoria.

```
function estrai(var p: puntatore; valore: integer): puntatore;
var corrente,precedente: puntatore;
begin
cerca_non_ordinata(p,valore,corrente,precedente);
  if corrente<>nil then
  begin
    if precedente<>nil then (* eliminazione in mezzo o in coda *)
      precedente^.pun:=corrente^.pun
    else    (* eliminazione in testa *)
      p:=corrente^.pun
  end;
  estrai:=corrente; (* eventualmente, nil *) end;
```

Ora che padroneggiate (J) i meccanismi di gestione delle liste siete in grado di capire gli svantaggi della gestione dinamica della RAM (dei vantaggi abbiamo già discusso...):

- Gli algoritmi sono più difficili ed è più facile commettere errori

- Non è possibile un accesso casuale agli elementi: per usare quello in posizione 'N' è necessario scorrere gli 'N-1' che precedono con un ciclo. Confrontate questa tecnica con l'immediatezza degli array: vett[N]

- A parità di tipo e di numero di elementi usa più memoria, quella usata per memorizzare i puntatori!! (ma in alcune situazioni globalmente abbiamo comunque un grosso risparmio: ricordate l'esercizio sullo SMAU e le 20.000 stringhe??)